University of California
Santa Barbara

Deduction-Powered Neural Program Synthesis: A Synergistic Perspective

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Yanju Chen

Committee in charge:

Professor Yu Feng, Chair
Professor Işıl Dillig
Professor Nadia Polikarpova
Professor Xifeng Yan

September 2023

The Dissertation of Yanju Chen is approved.

Professor Işıl Dillig

Professor Nadia Polikarpova

Professor Xifeng Yan

Professor Yu Feng, Committee Chair

July 2023

Deduction-Powered Neural Program Synthesis:

A Synergistic Perspective

by

Yanju Chen

Dedicated to my family,

for the days and nights we spent together

in memory, in dreams, and in spirit.

Acknowledgements

I have never stopped questioning myself for pursuing a Ph.D. — for hundreds of nights I wished I could convince myself of such a foolish mistake and give in — until my advisor, *Yu Feng*, picked me up from the desert that I had long immersed myself in, and showed me the million stars in the sky. From him I learned to imagine, to persist and to deconstruct, to discover, to compose and to restructure — science and research to me have never been so fulfilling and rewarding. I treasure the time we spent together focusing, brainstorming and arguing, and the determinism to succeed from his eyes has always been the strongest lighthouse in my eternal storms.

I wish I had been stronger, smarter and more confident at the very beginning; but obviously I'm not the chosen one, and it took time and a great ton of perseverance and resolution to make even a single step. I'm grateful that my dissertation committee has been supportive with me and providing valuable and professional feedback while I paid for those steps. The most insightful conversations are those with *Isıl Dillig* where words and sentences are articulated precisely to their meanings — I strive to be rigorous on science disciplines and methodologies just like she does, and it proves to be a great benefit[1]; *Nadia Polikarpova* is the first fan[2] of my work and she shows great passion for life and research that I always look for; *Xifeng Yan* is the shepherd that guided me through darkness and crises in both my physical and mental worlds.

The beautiful coastline of Santa Barbara has always been a great relief to any long-lasting pains and doubts — the perpetual waves understand and consume every piece of sorrow, return with noise, and bring it to the purple sunset to sink with every peaceful night. I found many answers here, with more confusions that I've learned to live with.

[1] Just like her cats that do college math well, it feels safe and comforting.

[2] I appreciate her first reply to my email with generous praise to my work; it had been, for a long time, the biggest support I ever received from the rest of the community.

The other calming hub from the campus is the PLSE lab, where I could usually find a snatch of joy and fun with my labmates besides a handful of equations and codes. I cherish the magical acquaintance in one of a million, and I'm grateful for the memory shared with friends of my life, folks from Veridise, 0xPARC, UT Austin and UW, and my mentors, collaborators and colleagues.

* * *

I've been having occasional flashbacks of the day I left my hometown for this journey six years ago, waving goodbye to my family members, when I was so arrogant and full of ignorance that I didn't even care to have a closer look at them, while they were still young and alive. I was such a fool thinking I'd be ready for everything — but obviously I am not. It was my parents that always stand behind me during the toughest time; they never complain and they save the best for me in their silent support. I wish I had known them better and felt things their way, so I'd be strong enough to see through every challenge ahead and stay true. Likewise, there are a thousand unspoken words behind a goodbye on every call with my grandparents that they would never let out in front of me, until they *thought* they had hanged up; I will never hear grandma's voice in tears because she lives with the stars now. This dissertation is dedicated to her: may this thread reach her.

Summer, 2023

Curriculum Vitæ
Yanju Chen

Education

2023	Ph.D. in Computer Science, University of California, Santa Barbara.
2017	M.S. in Computer Science, Sun Yat-sen University.
2014	B.S. in Computer Science, Sun Yat-sen University.

Publications

1. [**USENIX Security'24**] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. In *33rd USENIX Security Symposium (USENIX Security 24)*, PHILADELPHIA, PA, August 2024. USENIX Association
2. [**ASE'23**] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. Fast and reliable program synthesis via user interaction. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, New York, NY, USA, 2024. Association for Computing Machinery
3. [**PLDI'23**] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of Under-Constrained circuits in Zero-Knowledge proofs. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023
4. [**PLDI'23**] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. Conflict-Driven synthesis for layout engines. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023
5. [**ASE'22**] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery
6. [**OOPSLA'22**] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Işil Dillig. Synthesis-Powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022
7. [**OOPSLA'22**] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using Neural-Guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022
8. [**PLDI'22**] Yanju Chen, Xifeng Yan, and Yu Feng. Visualization question answering using introspective program synthesis. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 137–151, New York, NY, USA, 2022. Association for Computing Machinery

9. [**ASPLOS'22**] Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. Tree traversal synthesis using Domain-Specific symbolic compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1030–1042, New York, NY, USA, 2022. Association for Computing Machinery

10. [**S&P'22**] P Bose, D Das, Y Chen, Y Feng, C Kruegel, and G Vigna. SAILFISH: Vetting smart contract State-Inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1235–1252, Los Alamitos, CA, USA, May 2022. IEEE Computer Society

11. [**ASE'20**] B Mariano, Y Chen, Y Feng, S K Lahiri, and I Dillig. Demystifying loops in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–274, Los Alamitos, CA, USA, September 2020. IEEE Computer Society

12. [**CAV'20**] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using Deduction-Guided reinforcement learning. In Shuvendu K Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 587–610, Cham, 2020. Springer International Publishing

13. [**FSE'19**] Yanju Chen, Ruben Martins, and Yu Feng. Maximal Multi-Layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 602–612, New York, NY, USA, 2019. Association for Computing Machinery

14. [**VLDB'19**] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *Proceedings VLDB Endowment*, 12 (12):1914–1917, August 2019

15. [**AAAI'17**] Yanju Chen and Rong Pan. Automatic emphatic information extraction from aligned acoustic data and its application on sentence compression. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 3422–3428, San Francisco, California, USA, 2017. AAAI Press

Abstract

Deduction-Powered Neural Program Synthesis:

A Synergistic Perspective

by

Yanju Chen

Program synthesis has found its unique position in automated programming for both end-users and developers; it is changing the way that users code. Recent advances in deep learning and computer-aided reasoning for program synthesis have greatly pushed both techniques for an open range of domains, e.g., data analysis, high-performance computing, design and reasoning of complex systems, web3 security and science. While algorithms from these two paradigms may place different assumptions of the problem (e.g., modality of specification) and guarantees for the result (e.g., completeness), it is usually difficult for users to benefit simultaneously from both. In fact, feedback generated by statistical and logical reasoning algorithms are usually found useful for each other, but they are seldom integrated in a seamless way for program synthesis due to the aforementioned difference.

Motivated by these challenges, this dissertation presents a program synthesis framework that unifies the two paradigms of statistical and logical reasoning. Specifically, we address this problem by three aspects. We first describe a unified *interface* that encodes user-provided specification from multi-modalities into machine-readable constraints by a hybrid approach of reasoning. The framework's *core* infrastructure is then powered by deduction-guided reinforcement learning, a novel approach that seamlessly incorporated feedback from logical reasoning into statistical models. We further demonstrate the *extent* of the framework by a derived system for reasoning and refinement of deep learning

model's predictions.

We implement the proposed techniques in research prototypes, whose effectiveness is confirmed by a set of extensive evaluations. Our proposed framework also brings improvements for end-user programming via broaden expressiveness, enhanced explainability and natural interactivity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the growing power of computation, program synthesis is gradually changing the way that users code: it helps automate tedious tasks (e.g., Microsoft FlashFill [49]), complete code (e.g., OpenAI Codex [23]) and refactor code (e.g., IntelliJ [77]). While the philosophy behind program synthesis is not restricted to automation of tedious programming tasks, it is more about automation of problem solving and solution discovery, as seen in competitive programming (AlphaCode [61]), math theorem proving (Google HOList [7]), and more — a thought process broadly required by all sciences.

1.1 Overview

Given a high-level specification of user intent, modern program synthesizers perform some form of backtracking search to find a program that satisfies the specification [45, 43, 109]. However, due to the enormous size of the search space, synthesizers additionally use at least one of two other techniques, namely logical and statistical reasoning, to make this approach practical. While both logical and statistical reasoning have been shown to dramatically improve search efficiency, there are still key challenges for existing

approaches:

1. Synthesized programs may not fully match user intent due to the incomplete nature of a single specification. Additional specifications are needed for refinement of result.

2. The two modes of reasoning are not tightly combined. For example, feedback from logical reasoning is often useful but not leveraged by statistical reasoning.

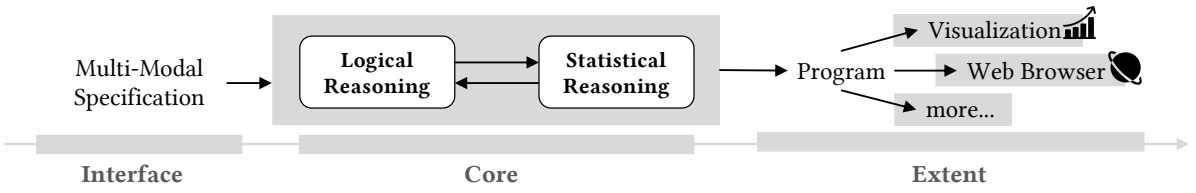


Figure 1.1: An overview of my dissertation research covering three aspects of a program synthesis framework.

This dissertation explores new paradigms that broaden the extent of program synthesis algorithms by tackling the aforementioned key challenges. Figure 1.1 shows an overview of my dissertation research covering three aspects of a program synthesis framework:

- The **interface** exploits the power of **multi-modal specification** that allows users to express their intent in multiple modalities, such as input-output examples, natural language descriptions, etc. We developed MARS [27], a synthesizer that can capture user intent beyond classical programming-by-example (PBE) tasks by encoding extra specification with statistical reasoning and decoding its output to guide the logical reasoning for synthesis.

- The **core** resides with a novel synthesis paradigm that tightly couples logical and statistical reasoning, denoted as **deduction-guided reinforcement learning**.

We developed CONCORD [28], a synthesizer that utilizes feedback from logical reasoning to improve the search performed by statistical reasoning. In particular, CONCORD frames a program synthesis problem as an instance of reinforcement learning, and extends the original learning algorithm to encode results from logical reasoning as additional training signals during search. The design of the core synthesis algorithm is flexible, in that it also allows improvement on top of statistical reasoning.

- The **extent** connects the interface and core with broader interdisciplinary scenarios to push for the boundary and imagination of program synthesis. We developed POE [31], a synthesizer that originates and extends from the synergistic bond established in CONCORD: it initiates the logical reasoning to refine the predictions from *off-the-shelf* statistical model, by synthesizing a program that *best* explains one or more of the predictions. The core insight of POE is to decipher a prediction by formalizing it as specification for a synthesis problem, thus exposing more information from the program synthesized that helps with the search.

This dissertation presents the above techniques under a unified framework, whose core philosophy resides in a synergistic bond between the two paradigms for program synthesis, namely statistical and logical reasoning. In what follows, we elaborate on the core aspects of this framework.

1.2 Multi-Modal Specification

Due to the *incomplete nature* of input-output examples, a synthesizer in a programming-by-example (PBE) task may generate programs that pass the examples but do not match the user intent. As a result, the user has to provide additional examples to refine the

results generated by the synthesizer, which imposes a huge burden to the user as it is tricky to: 1) figure out the root cause of the wrong candidates and 2) come up with better examples to refine the output of the synthesizer. In fact, on technical forums like Stackoverflow, a user typically would describe the problem with a combination of data from multiple modalities: input-output examples, natural language descriptions, partial code snippets, etc., which as a whole contributes to the user intent in a more thorough and accurate way. This motivates MARS [27], a synthesizer that is capable of capturing user intent in multi-modal specification.

The core of MARS consists of two folds: a logical reasoning engine that prunes program search space, and a statistical model that learns to prioritize search preference over promising programs. MARS captures specification of different types in different ways:

- For *hard* specification that the synthesized program must *always* satisfy, such as input-output examples, MARS encodes them directly as logical forms like existing approaches;
- For *soft* specification that the synthesized program should *try* to satisfy, such as natural language descriptions, MARS captures them using a statistical model (neural network) since they are mostly noisy. MARS then devises a set of logical predicates to describe the implication of the output of statistical model, which encodes the implication of soft specification.

As a result, MARS finds an optimal solution program by solving the problem with logical forms encoded from both types of specification, i.e., MARS finds programs that 1) satisfy all hard specification, and 2) satisfy the most soft specification. On a set of 80 challenging real-world data science tasks, MARS demonstrates the effectiveness of incorporating multi-modal specification by reducing more than 80% of the timeout cases of state-of-the-art tools and reaches an averaged 15× speedup for solved benchmarks.

Overall, MARS shows the power of multi-modal specification for enriching the interface of a typical program synthesis framework, which consolidates a broader spectrum of inputs for the core algorithms of the synthesis framework, and motivates the exploration on an in-depth connection between the two components mentioned above for logical and statistical reasoning.

1.3 Deduction-Guided Machine Learning

Existing synthesizers like MARS [27] contains two modes of reasoning, namely logical and statistical reasoning. Even though they are proven to be effective for program synthesis, they are not tightly coupled in existing synthesizers. In particular, feedback from logical reasoning is not leveraged by statistical models, which deviates from the intuition of human thought process, where deduction (logical reasoning) and perception (statistical reasoning) are coupled synergistically in problem solving.

Motivated by such observation, CONCORD [28] is developed to bridge the gap between the two modes by combining them in a synergistic way. Similar to prior techniques, CONCORD starts with a statistical model (henceforth called a policy) that is trained offline on a representative set of training problems and uses this policy to guide the search. However, unlike prior techniques, CONCORD updates this policy online at synthesis time and gradually improves the policy by incorporating feedback from a logical reasoning engine. Specifically, CONCORD formulates program synthesis as a reinforcement learning (RL) problem and devises a novel algorithm that converts feedback from logical reasoning into representative data that is then used for improving the policy. While RL proves to be a good fit for the problem, standard RL algorithms typically update the policy based on feedback received on search spaced that is *already* explored. However, in the context of program synthesis, logical reasoning can also provide feedback about search space that

has *not* been explored. On a set of 100 challenging list processing problems, CONCORD is proven effective in that it solves 15% more of them than state-of-the-art synthesizers with an average of 8.7× faster.

CONCORD's core algorithm is dubbed as *deduction-guided machine learning*. On one hand, CONCORD demonstrates the effectiveness of such approach when the user has access to the statistical model; on the other, such a tight bond also applies to more complex black-box deep learning models, on a synthesizer I develop called POE [31]. POE is motivated by visualization question answering (VQA) tasks, where a model is required to answer visualization queries from natural language descriptions. Since annotated data that contains both queries, logical forms and answers is very expensive, recent state-of-the-art deep learning models are trained only with queries and answers in exchange for more data and less manual efforts, which in return, are not always leading to satisfactory answers. To mitigate this problem, POE incorporates a synergistic procedure between the logical and statistical reasoning to refine potentially problematic predictions from the deep learning model. In particular, POE generates programs that are consistent via program synthesis with model predictions as specification, and decides the most promising prediction by selecting the one that best aligns with the problem context. The core insight behind POE is that, even though the statistical model generates a wrong answer that is derived from a sequence of hidden inference steps, part of them may still be sensible since training is based on a large corpus; by synthesizing the program that satisfies the prediction, more information is exposed to help make better predictions. As a result, POE's experiments demonstrate its effectiveness on a set of 629 real-world challenging VQA tasks, by a 15% improvement on the number of benchmarks solved than state-of-the-art methods. POE opens up brand new potentials for coupling the two modes of logical and statistical reasoning.

In summary, this dissertation makes the following key contributions:

- We design a customized deep neural network architecture for learning the user's preference using an aligned corpus that maps the user's textual information to the desired solutions. Based on this architecture, we design a novel multi-layer specification that allows the end-user to specify her intent using soft and hard constraints.

- We propose MARS, a Max-SMT based synthesis framework that takes as input a multi-layer specification and enumerates solutions that are close to the user's intent. Our framework is parameterized with the underlying neural networks and the DSL, which can be easily instantiated to different domains.

- We propose CONCORD, a synthesizer consisting of: 1) a new synthesis algorithm based on reinforcement learning that tightly couples statistical and deductive reasoning, and 2) an off-policy reinforcement learning technique that uses the output of the deduction engine to gradually improve its policy.

- 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 propose POE, a synthesizer equipped with: 1) an abstract program synthesis algorithm for quickly inducing the search space given *noisy* specifications from a deep learning model's output, and 2) an optimal program synthesis algorithm for finding programs that best match the consistency constraints implied between natural language questions and visualizations.

- Our end-to-end systems – MARS, CONCORD and POE– are empirically evaluated in different program synthesis domains to show the effectiveness of the proposed techniques and algorithms.

The rest of this dissertation is organized as follows:

- Chapter 2 describes a program synthesis framework MARS that utilizes multi-layer specification, where statistical outputs are encoded into logical reasoning.
- Chapter 3 presents a program synthesis algorithm CONCORD that connects deduction-based reasoning with machine learning.
- Chapter 4 extends the synergistic bond between statistical and logical reasoning into the POE framework that provides explanation and refinement for deep learning models.
- Chapter 5 discusses related work and Chapter 6 concludes.

Chapter 2

MARS: Program Synthesis Using Multi-Layer Specification

In today's data-centric world, data analytics has become one of the key elements in our daily life, including science, politics, business, and international relations. On the other hand, due to the messy nature of data in different application domains, data scientists spend close to 80% [36] of their time performing data wrangling tasks, which are considered to be the "janitor work" of data science.

To mitigate this problem, in recent years, there has been significant interest in end-user program synthesis for data science, in which the goal is to automate tedious data analytics tasks from informal specifications, such as input-output examples [49, 43] or natural language [113, 90]. For instance, programming-by-example (PBE) has been used to automate tedious tasks such as string manipulations in Excel [49], data wrangling tasks on tabular and hierarchical data [43, 112], and SQL queries [106]. Despite significant progress in PBE systems, expressing the user intent still remains a major challenge. As a result, due to the *incomplete nature* of input-output examples, a synthesizer may generate programs that pass the examples but do not match the user intent. In that case, the

user has to provide additional examples to refine the results generated by the synthesizer, which imposes a huge burden to the end-user as it is tricky to figure out the root cause of the wrong candidates [82] and come up with new examples to refine the output of the synthesizer.

To address the above limitation, this chapter aims to design a synthesis framework that *accurately* captures the user intent. By looking at hundreds of relevant data analytics questions from StackOverflow, we observe that an end-user typically describes her problem in *a combination of* input-output examples, natural language description, partial code snippet, etc. To give readers our insight, consider an example from StackOverflow in Figure 2.1. Here, the user has an input table and wants to transform it into an output table with a different shape. As shown in Figure 2.1, the correct solution (on the right) requires merging two column (i.e., `unite`), aggregating (i.e., `group_by`, `summarise`) the sum of another column, and finally pivoting (`spread`) the returning table. To solve this benchmark, it takes MORPHEUS [43], the state-of-the-art synthesizer for data wrangling tasks, around five minutes. Moreover, if the program found by MORPHEUS does not match the user intent, she has to refine the input-output examples and rerun the synthesizer.

In a lot of cases, the information provided by the end-user typically goes beyond input-output examples. In most helper forums (e.g., StackOverflow), we observed that people usually describe problems in the combination of natural language and input-output examples. For instance, looking at the example in Figure 2.1, the user not only provides input-output examples, but also indicates a rough "sketch" of the solution through natural language. For instance, the "reshape" and "count" keywords indicate that the solution should use library functions that perform pivoting (i.e. `spread` or `gather`) and aggregation (i.e., `group_by + summarise`), respectively. Other keywords such as "total found" suggest that `sum` should be used together with `summarise`, and the keyword "Sp_B_pos"

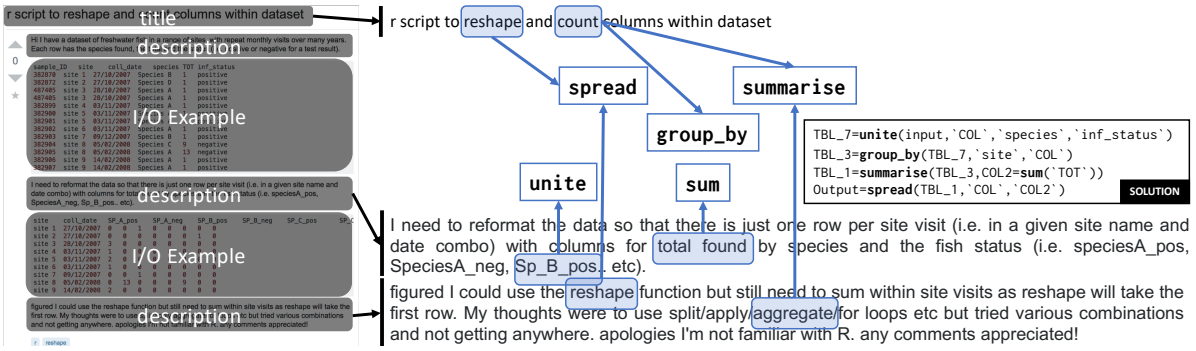


Figure 2.1: A motivating example from StackOverflow.[1]

that the function call `unite` should be used. If we use arrows to visualize the connection between text description and function calls from data-wrangling libraries, we can observe a strong connection between the user intent and the solution.

However, real world textual information is inherently noisy and ambiguous. As a result, it is very challenging to derive the right mapping from the textual information to their corresponding function calls. Second, even if we have the right mapping, it is still unclear how to integrate this information into most existing PBE systems [49, 10, 112, 106, 118], which typically rely on their efficient search algorithms by leveraging the syntax or semantics of the input-output examples.

We propose MARS, a novel synthesis framework that takes as input a *multi-layer specification* that appears in a large class of applications. Here a *multi-layer specification* is composed by input-output examples, textual description, and partial code snippets that express the user intent. To solve a multi-layer specification synthesis (MSS) problem, MARS encodes input-output examples as *hard constraints* which have to be satisfied, and denotes additional preferences (e.g., textual description, partial code snippet, etc) as *soft constraints* which are *preferably* satisfiable. After that, the MSS problem is reduced to the *maximum satisfiability modulo theories* (Max-SMT) problem which can be efficiently

[1] `https://stackoverflow.com/questions/39369502`

solved by an off-the-shelf SMT solver [37, 16]. The Max-SMT encoding of the MSS problem aims to satisfy the input-output constraints and maximize the user intent that is obtained from natural language, partial code snippet, and intermediate results.

To accurately capture the user intent from noisy and ambiguous description, we propose a hybrid neural architecture that combines the power of an LSTM-based sequence-to-sequence (i.e., *seq2seq*) [101] model and the *apriori algorithm* [1] for mining association rules. In particular, our *seq2seq* model encodes the probability of a *symbolic program* (i.e., a program of which constants are unknown.) given its corresponding textual description. However, like other deep learning applications, the performance of a *seq2seq* model heavily relies on the quality and quantity of the training data. Therefore, as shown in Section 2.6, for benchmarks of which solutions are complicated and rarely appear in the training set, our *seq2seq* model may not suggest the right candidates. To mitigate this problem, we leverage the *apriori algorithm* for mining the extra hidden information that can not be covered by the *seq2seq* model. Intuitively, through unsupervised learning, the *apriori algorithm* is used to mine association rules that indicate the hidden connections between words and individual functions. After that, we use the association rules for refining the original rankings of the *seq2seq* model.

To evaluate the effectiveness of our technique, we instantiate MARS into the data wrangling domain and compare it against MORPHEUS [43], the state-of-the-art PBE synthesizer for data wrangling tasks. We evaluate both approaches on the 80 benchmarks from MORPHEUS [46], and show that MARS outperforms MORPHEUS in terms of running time and number of benchmarks being solved. For challenging benchmarks, our approach is on average 15x faster than the MORPHEUS tool.

To summarize, this chapter focuses on the following key contributions:

- We design a customized deep neural network architecture for learning the user's

preference using an aligned corpus that maps the user's textual information to the desired solutions.

- We design a novel multi-layer specification that allows the end-user to specify her intent using soft and hard constraints.
- We propose a Max-SMT based synthesis framework that takes as input a multi-layer specification and enumerates solutions that are close to the user's intent. Our framework is parameterized with the underlying neural networks and the DSL, which can be easily instantiated to different domains.
- We integrate MARS's hybrid model into the MORPHEUS tool and empirically evaluate our approach in the data wrangling domain by showing that MARS outperforms the state of the art in running time and number of benchmarks solved.

2.1 Overview

In this section, we give an overview of our approach with the aid of the motivating example in Figure 2.1. Specifically, as shown in Figure 2.2, we use a simplified domain-specific language (DSL) based on `dplyr` and `tidyr`, which are two popular libraries for data wrangling tasks in *R*.

In this example, the user wants to perform a complex data wrangling task which requires concatenating two columns (i.e., `unite`), aggregation (i.e., `summarise`), and table pivoting (i.e., `spread`). We now explain the key ideas that enable MARS to solve this complex problem.We use abstract syntax trees (AST) to represent programs. For example, Figure 2.3 shows an AST that represents a *symbolic program* where some of the nodes are still unknown. A symbolic program can be instantiated in many ways and can generate several thousand concrete programs. For instance, the concrete program

$$
\begin{aligned}
T \rightarrow\ & x_i \mid \texttt{spread(T, COL,COL)} \mid \texttt{unite(T, COL, COL)} \\
& \texttt{group\_by(T, LIST)} \mid \texttt{summarise(T, AG, COL)} \\
& \texttt{gather(T, LIST)} \mid \texttt{select(T, LIST)} \\
\texttt{LIST} \rightarrow\ & \texttt{[1]} \mid \texttt{[1,2]} \mid \ldots \mid \texttt{[4,5]} \\
\texttt{COL} \rightarrow\ & \texttt{0} \mid \ldots \mid \texttt{10} \\
\texttt{AG} \rightarrow\ & \texttt{sum} \mid \texttt{mean} \mid \texttt{max} \mid \texttt{min}
\end{aligned}
$$

Figure 2.2: The grammar of a DSL for data wrangling tasks in `dplyr` and `tidyr`.

represented in Figure 2.4 corresponds to the following assignment:

$$\{N_1 \mapsto \texttt{select},\ N_2 \mapsto \texttt{gather},\ N_3 \mapsto \texttt{[1,2]},\ N_4 \mapsto x_0,\ N_5 \mapsto \texttt{[1,3]}\}$$

This approach, while being general, has several drawbacks. First, since input-output examples are imprecise specifications, a synthesizer may generate a candidate that does not match the *user intent*, which requires the user to provide additional examples to refine the result [106, 49]. Second, given a specific task, there can be many candidates satisfying the input-output examples but only few of them match the user intent. In this case, a synthesizer typically enumerates solutions according to some heuristic, such as the size of AST [47], or keywords provided by the user [106]. None of the previous work proposes a systematic solution for unifying the user intent from different sources.

MARS takes a different step by proposing a *multi-layer specification* that combines input-output examples with additional hints from the user. For instance, looking at the StackOverflow example in Figure 2.1, in addition to the input-output tables, the user also provides extra hints using natural language and intermediate results. Specifically, the word "reshape" in the title indicates that the solution should use either `spread` or `gather`, and "count" suggests the occurrence of aggregate functions(i.e., `summarise`, `group_by`).

To incorporate the additional information, we propose a novel *hybrid neural architecture* by leveraging the advantages of a seq2seq [101] model and the *apriori algorithm* for learning association rules [2]. In particular, the seq2seq model takes as input the text description and returns the most likely symbolic program according to a statistical model trained from a corpus. For the example in Figure 2.1, our `seq2seq` model suggests some of the following candidates:

```
{mutate,group_by,summarise,spread} (92)
{group_by,summarise,mutate,select} (91)
...
{unite,group_by,summarise,spread} (79)
...
```

Each item in the list is a pair $(\mathcal{P}, w_i)$ where $\mathcal{P}$ represents a symbolic program that we learn from the data, and w_i denotes the likelihood of being part of the solution. By leveraging the additional description from the user, the seq2seq model is able to suggest candidates that are close to the user intent. However, due to the size and quality of the training data, for complex solutions which *rarely appear* in the corpus, the seq2seq model is unlikely to suggest the correct symbolic program. As a result, a synthesizer may still spend a significant amount of time enumerating wrong candidates. For instance, by following the ranking generated from the seq2seq model, a synthesizer has to explore 130 symbolic programs before finding the right candidate.

To mitigate the above limitation, we leverage the *apriori* algorithm [1] for mining association rules. Intuitively, an association rule, which is learned from a corpus of data through unsupervised learning, aims to identify the hidden connections among the

keywords. For instance, given the text description in Figure 2.1, our algorithm is able to discover the following rule which suggests that `spread` has a high chance to appear in the solution:

$$\{\texttt{reshape}, \texttt{count}\} \Rightarrow \{\texttt{spread}\}$$

and the following rule indicates that `unite` should also appear in the solution:

$$\{\_, \texttt{reshape}\} \Rightarrow \{\texttt{unite}\}$$

Using our *refinement algorithm* discussed in Section 2.3.3, our system is able to incorporate the hints from the association rules to adjust the distribution of the seq2seq model. For instance, after running the refinement algorithm, the previous ranking is adjusted to:

...

$\{\texttt{unite}, \texttt{group\_by}, \texttt{summarise}, \texttt{spread}\}$ (109)

...

$\{\texttt{mutate}, \texttt{group\_by}, \texttt{summarise}, \texttt{spread}\}$ (96)

$\{\texttt{group\_by}, \texttt{summarise}, \texttt{mutate}, \texttt{select}\}$ (94)

Observe that the score of all three candidates get increased as they are connected to association rules learned from data. The score of the correct candidate increases more as this candidate matches *more rules* than others. As a result, a synthesizer only needs to explore less than 30 symbolic programs before reaching the right one.

To incorporate the above ranking from our statistical model, MARS provides *soft constraints* in the form of $(f(s_1, ..., s_k), w_i)$ where f is a k-nary predicate over DSL constructs

with likelihood weight w_i. For instance, the symbolic program of the correct candidate can be expressed with the following *soft constraints*:

$$\begin{aligned}
&\texttt{(occurs(unite), 109)} \wedge \texttt{(occurs(group_by), 109)} \wedge \\
&\texttt{(occurs(summarise), 109)} \wedge \texttt{(occurs(spread), 109)} \wedge \\
&\texttt{(hasChild(group_by,unite), 109)} \wedge \\
&\texttt{(hasChild(summarise,group_by), 109)} \wedge \\
&\texttt{(hasChild(spread,summarise), 109)}
\end{aligned}$$

Here, $\texttt{hasChild}(s_i,\ s_j)$ is a binary predicate which indicates that the DSL construct s_i should be the parent of s_j in the solution. Similarly, $\texttt{occurs}(s_i)$ is a unary predicate asserting that s_i should occur in the solution. Given the soft constraints generated by the hybrid model, the underlying Max-SMT solver in MARS can enumerate candidates in a way that $\sum \omega_i$ is maximized. In other words, MARS always prioritizes candidates that not only pass the input-output examples, but are also consistent with the user intent expressed in natural language.

2.2 Problem Formalization

This section proposes a general setting for our synthesis problem, and formally states the definitions of our multi-layer specification and maximal synthesis.

Given a domain-specific language (DSL) described by a context-free grammar $\mathcal{G}$, our synthesis framework searches the space of all possible programs up to a given depth.

A DSL is a tuple (Σ, R, S), where Σ, R, and S represent the set of symbols, productions, and the start symbol, respectively. Each symbol $\chi \in \Sigma$ corresponds to our built-in DSL construct (e.g., +, `spread, gather, select`, etc.), constants, and variables. Pro-

gram inputs are expressed as symbols $x_1, \ldots, x_k \in \Sigma$. Every production $p \in R$ has the form $p = (A \to \chi(A_1, \ldots, A_k))$, where $\chi \in \Sigma$ is a DSL construct and $A_1, \ldots, A_k \in \Sigma$ are symbols for the arguments. *Symbolic* and *concrete* programs are defined using symbols from the DSL.

Definition 2.2.1. (Symbolic Program) A symbolic program $\overline{\mathcal{P}}$ is an abstract syntax tree (AST) where some labels of the AST nodes are represented as *symbolic variables* yet to be determined.

Example 2.2.1. Figure 2.3 shows a symbolic program with depth of size two. Here, s_3, s_4, and s_5 denote symbolic variables which corresponds to unknown symbols. This symbolic program corresponds to `select(gather(?, ?), ?)`, where the `?` denotes symbolic variables that still need to be determined.

Intuitively, a symbolic program $\overline{\mathcal{P}}$ represents partial programs where some of the symbols are unknown. In Section 2.3, we will introduce a neural architecture for learning the most likely symbolic programs from a corpus of data.

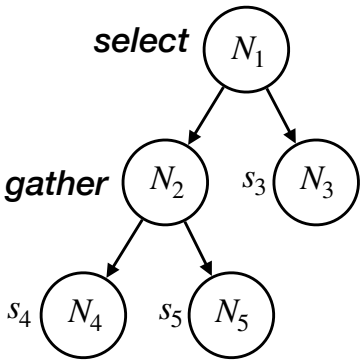


Figure 2.3: An example of symbolic program.

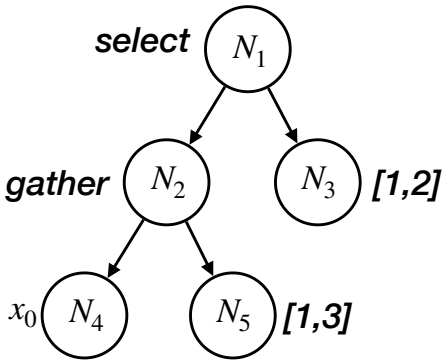


Figure 2.4: An example of concrete program.

Definition 2.2.2. (Concrete Program) A concrete program $\mathcal{P}$ is an AST where each node is labeled with a symbol from the DSL.

Example 2.2.2. Figure 2.4 shows an AST which corresponds to the concrete program: `select(gather(`x_0, [1,3]), [1,2]).

Definition 2.2.3. (Hard Specification) The hard specification expresses a set of constraints that the symbolic program $\overline{\mathcal{P}}$ has to satisfy. In classical PBE systems, we often refer to the input-output examples as the hard specification. In particular, $\overline{\mathcal{P}}(\mathcal{E}_{in}) = \mathcal{E}_{out}$.

Example 2.2.3. In MARS, the hard specification is used to encode the input-output requirement from the end-user. E.g., in Figure 2.1, the input and output tables are translated into hard constraints in MARS.

Definition 2.2.4. (Soft Specification) The soft specification denotes a set of constraints that the symbolic program $\overline{\mathcal{P}}$ *preferably* satisfies. In particular, each soft constraint is denoted by a pair $(pr(\chi_1, \ldots, \chi_k), \omega)$ where $pr(\chi_1, \ldots, \chi_k)$ is a k-ary predicate over the DSL constructs and ω represents the predicate confidence.

Example 2.2.4. In MARS, the soft specification is used to encode the user preference in the form of natural language. For instance, the unary predicate $(\texttt{occurs}(\chi_i), \omega_i)$ encodes that a DSL construct χ_i should appear in the program with confidence ω_i. Similarly, the binary predicate $(\texttt{hasChild}(\chi_i, \chi_j), \omega_j)$ denotes that a DSL construct χ_i should appear as the parent of χ_j in the program with confidence ω_j. Note that the weight of each predicate is automatically learned from a corpus of data.

Now we are ready to formally state our synthesis problem.

Definition 2.2.5. (Maximal Multi-layer Specification Synthesis) Given specification $(\mathcal{E}, \Psi, \Sigma)$ where $\mathcal{E} = (\mathsf{T}_{in}, \mathsf{T}_{out})$, $\Psi = \bigcup (\chi_i, \omega_i)$, and Σ represents all symbols in the DSL, the Maximal Multi-Specification Synthesis problem is to infer a program $\mathcal{P}$ such that:

- $\mathcal{P}$ is a well-typed expression over symbols in Σ.

- $\mathcal{P}(\mathsf{T}_{in}) = \mathsf{T}_{out}$.
- $\sum \omega_i$ is maximized.

2.3 Neural Architecture

In this section, we propose a hybrid neural architecture for inferring the most promising symbolic programs given the user description. In particular, our architecture incorporates a sequence-to-sequence (seq2seq) model and the *apriori* algorithm for discovering association rules through *unsupervised learning*. While the seq2seq model is for estimating the initial score of a symbolic program, the association rules are further used to adjust the initial score by mining hidden information that can not be identified by the seq2seq model.

2.3.1 Sequence-To-Sequence Model

The problem of inferring the most promising symbolic programs from user description can be viewed as a translation between two different languages. In particular, our goal is to translate from natural language to symbolic programs expressed in our DSL. Inspired by the recent success in natural language processing, we apply a seq2seq model with *Long Short-Term Memory (LSTM)* [52] cells.

As shown in Figure 2.5, given a *question-solution* pair (D, S), where a *question* is a user description composed by word tokens d:

$$D = (d_1, d_2, \ldots, d_n),$$

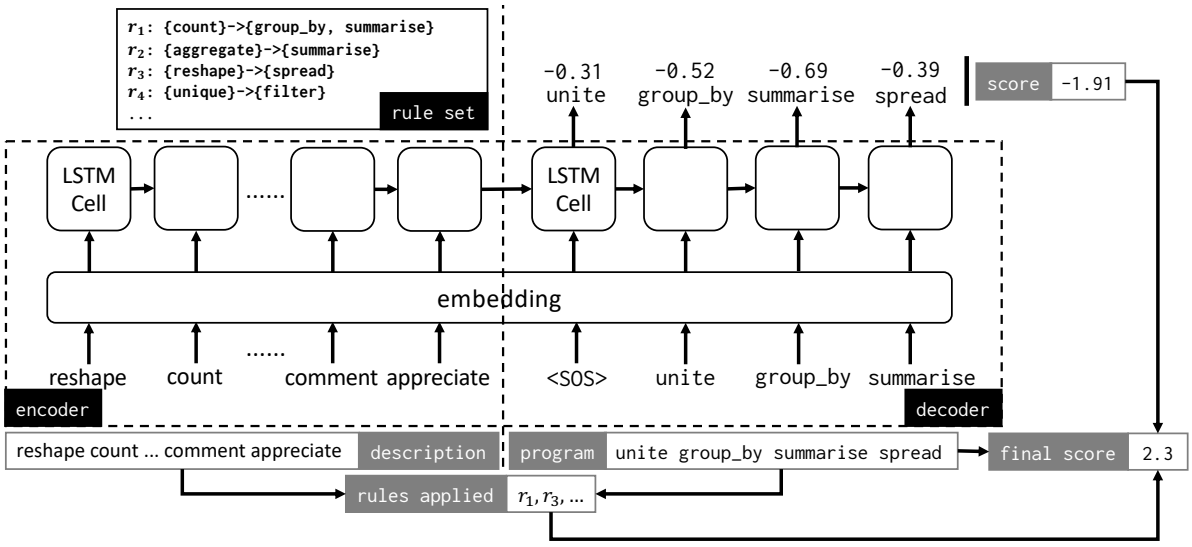


Figure 2.5: The hybrid neural architecture in MARS

and a *solution* is a symbolic program composed by a sequence of functions[2] s_i:

$$S = (s_1, s_2, \ldots, s_m),$$

the *seq2seq* model is used to estimate the probability of $P(S|D)$, which is then given by:

$$P(S|D) = P(s_1, s_2, \ldots, s_m | d_1, d_2, \ldots, d_n) = \prod_{t=1}^{m} P(s_t | v, s_1, s_2, \ldots, s_{t-1}),$$

where v is a fixed-dimensional vector representation of the user description generated by the *encoder*.

Internally, the seq2seq model is composed by two components: the *encoder* and the *decoder*. The *encoder* is an *LSTM* cell that takes as input a *question* D and generate its corresponding vector representation v. At every time step t, we feed each token d_t from the *question* to the *encoder* and compute the following functions as given by the *LSTM*

[2]Each symbolic program ignores all constant variables and only preserves the name of each function.

mechanism:

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, d_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, d_t]) \\
\tilde{h_t} &= \tanh(W \cdot [r_t * h_{t-1}, d_t]) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h_t},
\end{aligned}
$$

where at time step t, h_t is the hidden state, W_* are network parameters that will later be learned from data, $[,]$ is the vector concatenation operation, $\cdot$ is matrix multiplication, and σ (sigmoid) and tanh are both activation functions that are given by:

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1 + e^{-x}} \\
\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}
\end{aligned}
$$

The final vector representation of a *question* is given by the last hidden state: $v = h_n$.

Similar to the *encoder*, the *decoder* is also composed by an *LSTM* cell which takes as input a symbolic program represented by a sequence of functions. The output of the *decoder* is a distribution of functions given the current hidden state h_i:

$$
u_i = W_u \cdot h_i + b_u,
$$

where W_u and b_u are both learnable parameters, and the probability for a specific function (for example, the jth function) at time step i is estimated by:

$$
P(s_{i,j}) = P(s_{i,j}|v, s_1, s_2, ..., s_{i-1}) = \frac{\exp(u_{i,j})}{\sum_j \exp(u_{i,j})},
$$

where $u_{i,j}$ is the jth element of the vector.

Finally, we use the *back propagation* method with *negative log likelihood* loss to learn the parameters of the neural network. The probability of a *symbolic program* given a *question* is computed by estimating the product of the probability at each time step. We take logarithm of every time step to prevent underflow of the final result, which gives the equation of the probability score as follows:

$$P(S|D) = P(s_1, s_2, ..., s_m|d_1, d_2, ..., d_n) = \sum_{t=1}^{m} \log P(s_t|v, s_1, s_2, ..., s_{t-1}),$$

where the most promising symbolic programs have higher scores.

2.3.2 Learning Association Rules

As shown later in Section 2.6, due to the quality of the training data, our seq2seq model alone does not always achieve good performance. Specifically, for complex benchmarks of which solutions rarely appear in the training data, it is difficult for the seq2seq model to suggest the right candidates. On the other hand, even though the user cannot figure out the exact solution for her problem, she may still indicate partial information of the desired solution using some keywords or phrases. In order to discover hidden information that can not be inferred by the seq2seq model, we leverage the *apriori* algorithm to mine association rules that will later be used to adjust the rankings from the seq2seq model.

As shown in Figure 2.5, let Q be the union of all tokens that appear in the questions and all functions that appear in a *solution*:

$$Q = q_1, q_2, ..., q_c,$$

and let E be the set of all tokens in a *question-solution* pair (S, D):

$$E = \bigcup(s_i, d_j) \text{ where } s_i \in S, d_j \in D$$

An *association rule* r of a given set E is defined by:

$$r : X \Rightarrow Y,$$

where $X, Y \subseteq Q$. For example,

$$\{\text{unite}, \text{wide}\} \Rightarrow \{\texttt{spread}\}$$

indicates that if the two keywords "unite" and "wide" appear in the *question*, then the function `spread` is also appearing in the corresponding *solution*. Also, rules can apply on functions:

$$\{\texttt{filter}, \texttt{summarise}\} \Rightarrow \{\texttt{group\_by}\}$$

which means if both `filter` and `summarise` appear in the *solution*, then the function `group_by` also appears in the same *solution*.

To learn the association rules, we run the *apriori* algorithm on more than 30,000 answers[3] from Stackoverflow. Since the *apriori* algorithm is based on *unsupervised learning*, it may generate rules that are not useful. To address this issue, we further filter out the

[3]An answer towards a specific question is usually composed by some natural language description and solution code, which fits the prerequisits of association rules mining.

association rules of which confidence are low according to the following formulas:

$$\mathsf{supp}(X) = \frac{|\{e \in E, X \subseteq e\}|}{|E|}$$
$$\mathsf{conf}(X \Rightarrow Y) = \frac{\mathsf{supp}(X \cup Y)}{\mathsf{supp}(X)}.$$

Here, supp indicates the frequency of X that appears in the dataset, and conf represents how often the rule holds.

2.3.3 Score Refinement Algorithm

In this section, we describe an algorithm that refines the score of the seq2seq model using the association rules in Section 2.3.2.

As shown in Algorithm 1, the key idea of our REFINEMENT procedure is to take as input a symbolic program S together with its original score c from the seq2seq model, and produce a new score c_r according to the association rules R discussed in Section 2.3.2. Internally, the refined score c_r is computed based on an *accumulative boosting ratio* b that is initialized at line 4. Then for each association rule r_i, the algorithm updates the *accumulative boosting ratio* based on a weight function θ as well as a *match* function that decides whether the current rule $r_i = X \Rightarrow Y$ applies to the current symbolic program together with its description (D, S):

$$\text{match}(r, D, S) = \begin{cases} 1 & \forall e \in X \cup Y, e \in D \text{ or } e \in S \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the weight function θ is used to measure the quality of association rule r_i by taking several factors into account, including the confidence (i.e. conf) and support (i.e., supp) discussed in Section 2.3.2, number of keywords that appear in rule r_i, and

cost of the DSL construct (e.g. compared to `select`, `mutate` is more computationally intensive).

Algorithm 1 Symbolic Program Score Refinement Algorithm

1: **procedure** REFINEMENT(R, D, S, c, θ)
2: **input:** association rule set R, *question* D, *solution* S with its corresponding score c and weight function θ
3: **output:** refined score c_r
4: $b \leftarrow 0$ ▷ accumulative boosting ratio
5: **for** rule $r_i \in R$ **do**
6: $b \leftarrow b + \theta(r_i) \cdot \text{match}(r_i, D, S)$
7: $c_r \leftarrow c + b \cdot |c|$ ▷ update score
8: **return** c_r

2.4 Maximal Specification Synthesis

In this section, we describe how MARS leverages the statistical information (discussed in Section 2.3) to enumerate programs that are close to user intent.

As we mentioned earlier, most PBE synthesizers [49, 10, 43, 112, 106, 118] perform program enumeration until they find a program that satisfies the input-output examples provided by the user.

In order to perform program enumeration, we first need to represent the set of all possible programs up to a given depth. Consider a DSL $D = (\Sigma, R, S)$ where Σ_m represent DSL constructs with arity-m and m is the greatest arity between DSL constructs. A symbolic program $\overline{\mathcal{P}}$ represented by a tree of depth k where each node has exactly m children can represent all programs that use at most $k-1$ production rules. Figure 2.6 shows a 3-ary tree with depth 2 that represents all programs that can be constructed using at most 1 production rule from the DSL shown in Figure 2.2. Note that $m = 3$ since the greatest arity between DSL constructs is 3.

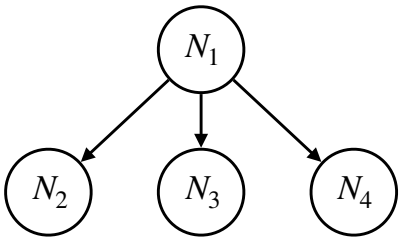


Figure 2.6: An example of a bounded symbolic program.

Example 2.4.1. Assigning $N_1 \mapsto \mathsf{unite}$, $N_2 \mapsto \mathsf{input}$, $N_3 \mapsto 1$, $N_4 \mapsto 2$ corresponds to the program "*unite(input, 1, 2)*" which unites columns 1 and 2 from table *input*.

Given a symbolic program $\overline{\mathcal{P}}$ and a DSL D, we encode the set of all possible concrete programs as an SMT formula φ. The Satisfiability Modulo Theories (SMT) problem is a decision problem for formulas that are composed with multiple theories. To encode symbolic programs, we use the quantifier free fragment of the theory of Linear Integer Arithmetic (LIA). A model of φ can be mapped to a concrete program by assigning a symbol to each node in $\overline{\mathcal{P}}$.

Variables For each node N_i, we use an integer variable with domain between 0 and r, where $r = |\Sigma|$. Assigning $N_i \mapsto k$ means that we assign to N_i the corresponding symbol. Let $\mathsf{idx} : \Sigma \to \mathbb{N}_0$ be a mapping between a symbol and its position. Since some production rules p may have arity smaller than m, there may exist some children nodes N_j that are not assigned any symbols. To enforce the invariant that each node is assigned *exactly one* symbol, we introduce a special symbol p_e with index 0 that is assigned to nodes without symbols, i.e. $N_j \mapsto 0$.

Example 2.4.2. Consider the DSL in Figure 2.2. idx maps each symbol to a corresponding integer that identifies its position. For example the input x_i is mapped to index 1,

spread to index 2, unite to index 3, etc.

Constraints Let I, O correspond to all symbols that are consistent with the input and output examples, respectively. To guarantee that all models correspond to well-typed concrete programs we must enforce the following constraints.

1. The root node N_1 of $\overline{\mathcal{P}}$ will be assigned a symbol that is consistent with the output type:

$$\bigvee_{p \in O} N_1 = \mathsf{idx}(p).$$

Example 2.4.3. Let $O = \{x_i, \text{ spread}, \text{ unite}, \text{ group\_by}, \text{ summarise}, \text{gather}, \text{select}\}$. The following constraint enforces that the output type is consistent with the output example:

$$N_1 = \mathsf{idx}(x_i) \vee N_1 = \mathsf{idx}(\mathsf{spread}) \vee N_1 = \mathsf{idx}(\mathsf{unite}) \vee N_1 = \mathsf{idx}(\mathsf{group\_by}) \vee$$
$$N_1 = \mathsf{idx}(\mathsf{summarise}) \vee N_1 = \mathsf{idx}(\mathsf{gather}) \vee N_1 = \mathsf{idx}(\mathsf{select}).$$

2. Let N be the set of all nodes and Ch_{N_i} the set of children nodes of $N_i \in N$. Furthermore, let $C(p, N_i)$ be the set of production rules that are consistent with production p and can be assigned to N_i. If a production rule $p = (A \to \chi(A_1, \ldots, A_k))$ is assigned to node N_i then all m children $N_j, \ldots, N_{j+m}$ will have to be consistent with $A_1, \ldots, A_k$:

$$\bigwedge_{p \in \Sigma, N_i \in N} N_i = \mathsf{idx}(p) \implies \bigwedge_{N_j \in Ch_{N_i}} \bigvee_{p_j \in C(p, N_j)} N_j = \mathsf{idx}(p_j).$$

Example 2.4.4. To guarantee that if production $p = \mathsf{unite}$ is assigned to node N_1 then

its children are consistent with p, we add the following constraints to φ:

$$N_1 = \mathsf{idx}(\mathsf{unite}) \implies \big(N_2 = \mathsf{idx}(x_i) \vee N_2 = \mathsf{idx}(\mathsf{spread}) \vee N_2 = \mathsf{idx}(\mathsf{unite}) \vee N_2 = \mathsf{idx}(\mathsf{group\_by}) \vee N_2 = \mathsf{idx}(\mathsf{summarise}) \vee N_2 = \mathsf{idx}(\mathsf{gather}) \vee N_2 = \mathsf{idx}(\mathsf{select})\big).$$

Similar constraints are added to guarantee the consistency of N_3 and N_4 when unite is assigned to N_1.

3. Let L the set of leaf nodes and T the set of terminal symbols. Only terminal symbols can be assigned to a leaf node:

$$\bigwedge_{N_i \in L} \bigvee_{p \in T} N_i = \mathsf{idx}(p).$$

Example 2.4.5. Consider the leaf node N_2. To restrict the occurrence of terminals in N_2, we add the following constraints:

$$N_2 = \mathsf{idx}(x_i) \vee N_2 = \mathsf{idx}([1]) \vee N_2 = \mathsf{idx}([1,2]) \vee \ldots \vee N_2 = \mathsf{idx}([4,5]) \vee N_2 = \mathsf{idx}(0) \vee \ldots \vee N_2 = \mathsf{idx}(10).$$

2.4.1 Enumerating Maximal Programs

Enumerating models from the SMT formula φ described in Section 2.4 will correspond to concrete programs. However, this enumeration does not take into consideration the user intent captured by the neural network described in Section 2.3. To capture this information, we extend the SMT formula to a Max-SMT (Maximum Satisfiability Modulo Theories) formula. A Max-SMT formula is composed by a set of *hard* and *soft*

constraints. The Max-SMT problem is to satisfy all hard constraints while maximizing the number of soft constraints that can be simultaneously satisfied. This problem can be further generalized to the weighted Max-SMT problem where each soft constraint c_i can be associated a weight w_i. As hard constraints, we use the constraints described in Section 2.4 that guarantee all enumerated programs are well-typed. As soft constraints, we use the predicates occurs and hasChild encoded as follows.

1. Let predicate $(\mathsf{occurs}(p_i), w_i)$ denote that a production rule p_i occurs with likelihood w_i in the final program. This predicate can be encoded into Max-SMT with the following soft constraints with weight w_i.

$$\bigwedge_{p_i \in \Lambda} \bigvee_{N_i \in N} N_i = \mathsf{idx}(p_i)$$

Example 2.4.6. The predicate $(\mathsf{occurs}(\texttt{spread}), 80)$ is encoded by adding the following soft constraint to φ with weight 80:

$$N_1 = \mathsf{idx}(\texttt{spread}) \vee N_2 = \mathsf{idx}(\texttt{spread}) \vee N_3 = \mathsf{idx}(\texttt{spread}) \vee N_4 = \mathsf{idx}(\texttt{spread}).$$

2. Let predicate $(\mathsf{hasChild}(p_i, p_j), w_i)$ denote that production p_i has production p_j as its children with likelihood w_i. This predicate is encoded as follows where all soft constraints have weight w_i.

$$\bigwedge_{p_i, p_j \in \Lambda, N_i \in N} N_i = \mathsf{idx}(p_i) \implies \bigwedge_{N_j \in Ch_{N_i}} N_j = \mathsf{idx}(p_j)$$

Example 2.4.7. The predicate $(\mathsf{hasChild}(\texttt{summarise}, \texttt{group\_by}), 92)$ is encoded by adding

the following constraints to φ with weight 92:

$$\big(N_1 = \mathsf{idx}(\texttt{summarise}) \implies N_2 = \mathsf{idx}(\texttt{group\_by})\big) \wedge$$
$$\big(N_1 = \mathsf{idx}(\texttt{summarise}) \implies N_3 = \mathsf{idx}(\texttt{group\_by})\big) \wedge$$
$$\big(N_1 = \mathsf{idx}(\texttt{summarise}) \implies N_4 = \mathsf{idx}(\texttt{group\_by})\big)$$

Maximizing the satisfaction of these soft constraints will guarantee that we enumerate programs that are closer to the user intent. Note that even though the predicates occurs and hasChild suffice to capture the information extracted by the neural network, our approach is not limited to these predicates and can be extended by adding additional predicates (e.g., happens before).

2.5 Implementation

Data Collection and Preparation We collect 20,640 pages from Stackoverflow [100] using the search keywords "tidyr" and "dplyr" (with testing benchmarks excluded), where each page contains a single *question* and multiple *solutions*. By removing duplicate contents and *questions* with no *solutions*, we obtain 16,459 *question-solution* pairs. Each *question* is pre-processed by a standard NLP pipeline that includes: stop word removal, lemmatization and tokenization, and a *solution* is represented as a sequence of DSL constructs (i.e., function names). The *question-solution* pairs are then used to train a seq2seq model. For the association rules mining, we extract *descriptions* from answers and their corresponding *solutions* and totally obtain 37,748 transactions as the input to the *Apriori* algorithm. To ensure the validity of our experiments, we remove all the benchmarks from the collected data.

Neural Network and Hybrid Architecture We build a seq2seq neural network using the PyTorch framework [81]. The hyper parameters (e.g., numbers of dimensions of the word/function embedding layer and LSTM hidden layer) are obtained through a simple grid search. For the seq2seq model in MARS, we set both the dimensions of word/function embedding layer and LSTM hidden layer to be 256, where the embedding layer maps 25,004 words and 14 functions[4] to vectors of the dimension 256. Furthermore, a single layer perceptron is connected to the hidden layer of each output time step in the decoder, mapping from a dimension of 512[5] to 14, which is used to predict the probability of each function given the previous hidden state and the current input.

As for the association rule mining, we apply the *Efficient-Apriori* [41] package to discover useful association rules that can be further applied to refine the original ranking generated by the *seq2seq* model. We then select *valid* rules according to the following criteria:

- confidence ≥ 0.9 or support ≥ 0.003.
- Each *valid* rule should have at least 1 *word* and 1 *function*. And the number of *functions* in the rules shall not exceed 2.
- Each *valid* rule should not contain any *stop* words, which builds upon the English stop words and includes additional *words* and *functions* that we consider less indicative.

By filtering out less relevant rules, we obtain 187 association rules.

[4]There are 25,000 natural language words in the word vocabulary and 10 functions in the function vocabulary. Each vocabulary contains 4 special helper tokens, namely namely "<PAD>" (empty placeholder), "<SOS>" and "<EOS>" (the start and end of a sequence), "<UKN>" (out-of-vocabulary word).

[5]Since we are using separate *seq2seq* structures for *title* and *question*, the concatenation of the hidden layers from both are of a dimension of 256*2=512.

Machine Configuration We train our seq2seq model on a machine from Google Cloud Platform with a 2.20GHz Intel Xeon CPU and an NVIDIA Tesla K80 GPU. All synthesis tasks were run on a laptop equipped with Intel Core i5 CPU and 16GB memory. Since the MORPHEUS tool is only available on a virtual machine [46], we used this virtual machine to run all program synthesis experiments. It took around 8 hours to train our hybrid model.

2.6 Evaluation

We evaluated MARS by conducting experiments that are designed to answer the following questions:

- Q1: Do our multi-layer specification and neural architecture suggest candidates that are close to the user intent?
- Q2: What is the impact of the neural architecture in MARS on the performance of a state-of-the-art synthesizer for data wrangling tasks?
- Q3: How is the performance of MARS affected by the quality of the corpus?

2.6.1 Quality of suggested candidates

To evaluate the benefit of the multi-layer specification and neural architecture in MARS, we instantiate the tool to the data wrangling domain, where data scientists tend to spend about 80% of their time doing tedious and repetitive tasks. In particular, we use the data in Section 2.5 to train the n-gram model from MORPHEUS [43], the seq2seq model discussed in Section 2.3.1, and the hybrid neural architecture described in Figure 2.5. Since the output of each model is a distribution of symbolic programs, we run all three models on the original benchmarks from MORPHEUS, which contains 80 data

wrangling tasks using two popular R libraries, namely, `tidyr` and `dplyr`. In particular, the data wrangling DSL contains 60 production rules and can induce a gigantic search space of the symbolic programs, posing a challenge for state-of-the-art synthesizers. As shown in MORPHEUS' user study, data scientists solved on average two benchmarks in one hour. For each benchmark, we then use the seq2seq model and hybrid neural architecture to enumerate symbolic programs and record the ranking of the correct candidate that matches the user intent. Finally, we manually checked all solutions synthesized by MARS and made sure that they are semantically equivalent to the reference solutions. Because the n-gram model in MORPHEUS only considers programs in the posts on StackOverflow and ignore user description, it provides a global ranking shared by all benchmarks.

Results As shown in Table 2.1, the average ranking and standard deviation of the n-gram model are 42 and 70, respectively. In other words, a synthesizer would need to explore 42 symbolic programs on average. Recall that a symbolic program may correspond to several thousands concrete programs. The standard deviation is used to quantify stability of the model. In contrast, by incorporating the user descriptions, the seq2seq model achieves an average ranking of 25 and a standard deviation of 39. Finally, with the help of the association rules, the hybrid model obtains the best performance with an average ranking of 18 and a standard deviation of 26. The result shows that our hybrid model not only suggests candidates that are close to user intent (i.e., low average), and it is also more stable (i.e., low standard deviation) across different benchmarks.

We further look into the number of top-1 and top-3 candidates that are correctly suggested by each model. As shown in Table 2.2, without user descriptions, the n-gram model fails to predict any correct candidates in top-1 and only suggests correct candidates in top-3 for two benchmarks. By leveraging user descriptions, the seq2seq model is able to figure out the right top-1 and top-3 candidates for 8 and 18 benchmarks, respectively.

Table 2.1: Statistics for different model rankings

model	*n-gram*	*seq2seq*	*hybrid*
average[*]	42	25	18
std.[†]	70	39	26

[1] standard deviation.
[*] computed based on the rankings of the correct solutions.

Table 2.2: Counts of top-1s and top-3s in different models

model	*n-gram*	*seq2seq*	*hybrid*
Top-1 total[*]	0	8	11
Top-3 total[*]	2	18	29

[*] computed based on the rankings of the correct solutions.

Finally, our hybrid model successfully suggests top-1 and top-3 candidates for 11 and 29 benchmarks.

2.6.2 Effectiveness of hybrid neural architecture

In this section, we further investigate the impact of a better ranking on the end-to-end performance of a synthesizer. Specifically, we integrate the previous three statistical models into MORPHEUS, a state-of-the-art synthesizer for data wrangling tasks.

Table 2.3: Statistics of running time

model	avg. speedup[1]	#timeouts[*]
ngram	1x	11
seq2seq	6x	8
hybrid	15x	2

[1] average speedup on challenging solved benchmarks.
[*] number of timeouts on all benchmarks.

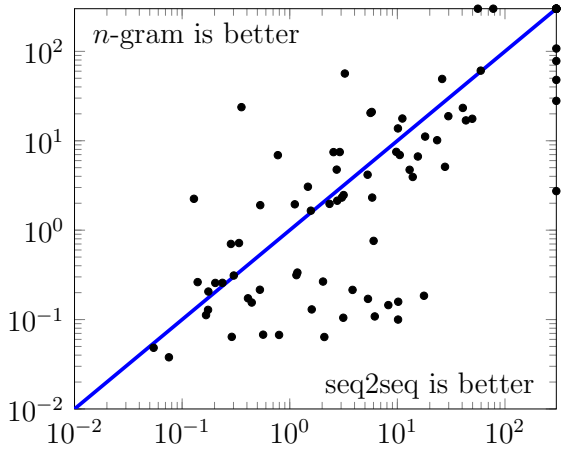


Figure 2.7: Comparison of run times (in seconds) between n-gram (x-axis, used in MORPHEUS) and seq2seq (y-axis, used in MARS) using a logarithmic scale.

Results Figure 2.7 and Figure 2.8 show the results of running MORPHEUS on its original 80 benchmarks with three different models (namely n-gram model in original MORPHEUS and seq2seq/hybrid model in MARS) and a time limit of 300 seconds. In particular, each dot in the figure represents the pairwise running time of a specific benchmark under different models. As a result, the dots near the diagonal indicates that the performance of two models is similar on those benchmarks. For instance, Figure 2.8 shows the comparison between the n-gram model and our hybrid model in terms of running time. Specifically, our hybrid model outperforms MORPHEUS' original n-gram model in 58 of 80 benchmarks. In the meantime, MORPHEUS times out on 11 benchmarks with the n-gram model, whereas it only times out on 2 benchmarks with our hybrid model. The performance of the seq2seq model is between the above two models by outperforming MORPHEUS n-gram model in 47 of 80 benchmarks and timing out on 8 benchmarks. Table 2.3 shows the average speedup for challenging benchmarks (i.e. > 3 library calls) with respect to the n-gram model for benchmarks that can be solved by both models. On average, the seq2seq model is 6x faster than the n-gram model and the hybrid model is 15x faster than the n-gram model. The result further confirms that a statistical model

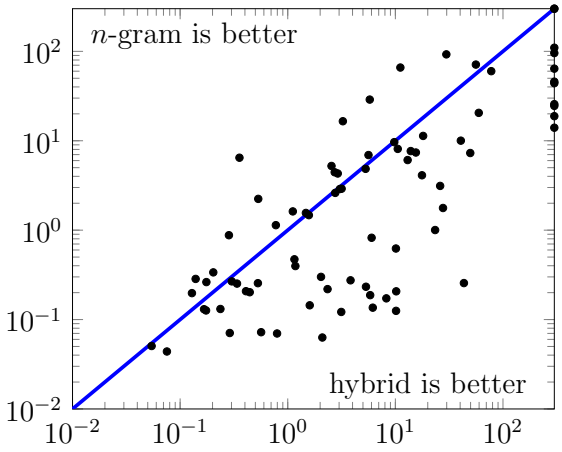


Figure 2.8: Comparison of run times (in seconds) between n-gram (x-axis, used in MORPHEUS) and hybrid (y-axis, used in MARS) using a logarithmic scale.

that accurately captures user intent tends to have a better performance in running time.

Remarks To understand the cases where our technique runs significantly faster, we manually look into some of the benchmarks. We notice that our technique performs especially well if the user states her problem in a clear way. For instance, in this post from StackOverflow, [6] although the user does not know the exact solution for her complex task, she is still able to convey the transformations using keywords (e.g., "count" and "unique") and partial code snippets. Even with these discrete signals, our hybrid model manages to guide MORPHEUS to the correct program in less than a second:

```
1 TBL_7 = filter(p25_input1, "b">1)
2 TBL_3 = unite(TBL_7, key_ab, "a", "b")
3 TBL_1 = group_by(TBL_3, "key_ab")
4 morpheus = summarise(TBL_1, e=n())
```

In contrast, MORPHEUS with its original n-gram model takes several minutes to find the right candidate.

[6] https://stackoverflow.com/questions/33549927

2.6.3 Discussion

Like any other technique, our approach also has its own limitations. For instance, in Figure 2.8, there are still some benchmarks where n-gram performs better, we manually inspect all these cases and notice that the issue is caused by the following reasons:

Insufficient Text In this post, [7] the user only provides input-output examples but her description barely contains any useful signals that allow our hybrid model to make a good prediction.

Contextual Text In this post, [8] the user explicitly states that she does not want to use the `mutate` function:

> *"... I can solve my problem using dplyr's* **mutate but** *it's a time-intensive, roundabout way to achieve my goal. ..."*

However, after tokenizing the natural description and removing all the stop words (e.g., *"but"*), our hybrid model loses the contextual information and takes `mutate` as the keyword.

Misleading Text In contrast to the previous example, in this post, [9] the user explicitly wants to use the `mutate` function:

> *"... I* **want to use mutate** *to make variable d which is mean of a,b and c. ..."*

However, since we directly adopt the DSL from MORPHEUS and the DSL does not support this special usage of `mutate`, our hybrid model proposes candidates that do not lead to the correct solution.

[7] `https://stackoverflow.com/questions/26733449`
[8] `https://stackoverflow.com/questions/29447325`
[9] `https://stackoverflow.com/questions/33401788`

2.6.4 Threats to Validity

Corpus Quality Even though the hybrid neural architecture is more resilient to the limitation of the existing data set, the performance of MARS is still sensitive to the quality of the training data. To mitigate this concern, we train our statistical model using all relevant posts from StackOverflow. In the future, we also plan to leverage transfer learning to incorporate resources written in other languages (e.g., Python and Matlab).

Benchmark Selection Due to the expressiveness of the DSL, in terms of complexity, the benchmarks from MORPHEUS [43] may not represent the actual distribution of the questions on StackOverflow. While the comparison on the MORPHEUS benchmarks may not completely unveil the benefit of our hybrid neural architecture, and a representative test suite may provide a more comprehensive view, we believe our comparison is sufficient to show the strength of our technique. Furthermore, since both our neural architecture and the enumerator are designed in domain-agnostic way, we also believe our technique can generalize to other domains.

2.7 Summary

In this chapter, we proposed MARS, a novel synthesis framework that takes as input a *multi-layer* specification which combines input-output examples, textual description, and partial code snippets to capture the user intent. To solve a multi-layer specification synthesis (MSS) problem, MARS encodes input-output examples as *hard constraints* and denotes additional preferences (e.g., textual description, partial code snippet, etc) as *soft constraints*. The MSS problem is reduced to a Max-SMT formula which can be solved by an off-the-self solver [37, 16]. To accurately capture user intent from noisy and ambiguous

descriptions, we propose a novel hybrid neural architecture that combines the power of a sequence-to-sequence model and the *apriori algorithm* for mining association rules. We instantiate our hybrid model to the data wrangling domain and compare its performance against MORPHEUS on its original 80 benchmarks. Our results show that our approach outperforms MORPHEUS and it is on average 15x faster for challenging benchmarks.

Chapter 3

CONCORD: Program Synthesis Using Deduction-Guided Reinforcement Learning

Due to its potential to significantly improve both programmer productivity and software correctness, *automated program synthesis* has gained enormous popularity over the last decade. Given a high-level specification of user intent, most modern synthesizers perform some form of backtracking search in order to find a program that satisfies the specification. However, due to the enormous size of the search space, synthesizers additionally use at least one of two other techniques, namely deduction and statistical reasoning, to make this approach practical. For example, many recent synthesis techniques use lightweight program analysis or logical reasoning to significantly prune the search space [45, 109, 43, 85]. On the other hand, several recent approaches utilize a statistical model (trained off-line) to bias the search towards programs that are more likely to satisfy the specification [6, 43, 12, 20]. While both deductive and statistical reasoning have been shown to dramatically improve synthesis efficiency, a key limitation of existing

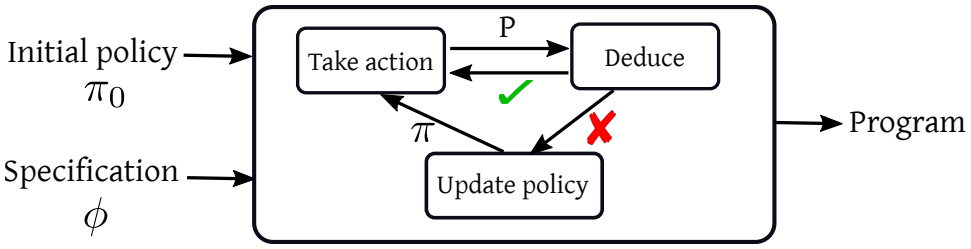


Figure 3.1: Overview of our synthesis algorithm

approaches is that they do not tightly combine these two modes of reasoning. In particular, although logical reasoning often provides very useful feedback at synthesis time, existing synthesis algorithms do not leverage such feedback to improve their statistical model.

We propose a new synthesis algorithm that meaningfully combines deductive and statistical reasoning. Similar to prior techniques, our approach starts with a statistical model (henceforth called a *policy*) that is trained off-line on a representative set of training problems and uses this policy to guide search. However, unlike prior techniques, our method *updates* this policy on-line at synthesis time and gradually improves the policy by incorporating feedback from a deduction engine.

To achieve this tight coupling between deductive and statistical reasoning, we formulate syntax-guided synthesis as a reinforcement learning (RL) problem. Specifically, given a context-free grammar for the underlying DSL, we think of partial (i.e., incomplete) programs in this DSL as states in a Markov Decision Process (MDP) and actions as grammar productions. Thus, a *policy* of this MDP specifies how a partial program should be extended to obtain a more specific program. Then, the goal of our reinforcement learning problem is to improve this policy over time as some partial programs are proven infeasible by an underlying deduction engine.

While the framework of reinforcement learning is a good fit for our problem, standard RL algorithms (e.g., policy gradient) typically update the policy based on feedback re-

ceived from states that have *already* been explored. However, in the context of program synthesis, deductive reasoning can also provide feedback about states that have *not* been explored. For example, given a partial program that is infeasible, one can analyze the root cause of failure to infer *other* infeasible programs [45, 110]. To deal with this difficulty, we propose an *off-policy* reinforcement learning algorithm that can improve the policy based on such additional feedback from the deduction engine.

As shown schematically in Figure 3.1, our synthesis algorithm consists of three conceptual elements, indicated as "Take action", "Deduce", and "Update policy". Given the current policy π and partial program P, "Take action" uses π to expand P into a more complete program P'. Then, "Deduce" employs existing deductive reasoning techniques (e.g., NEO [45], TRINITY [69]) to check whether P' is feasible with respect to the specification. If this is not the case, "Update policy" uses the feedback provided by the deduction engine to improve π. Specifically, the policy is updated using an off-policy variant of the *policy gradient* algorithm, where the gradient computation is adapted to our unique setting.

We have implemented the proposed method in a new synthesis tool called CONCORD and empirically evaluate it on synthesis tasks used in prior work [45, 6]. We also compare our method with several relevant baselines as well as two existing synthesis tools. Notably, our evaluation shows that CONCORD can solve 15% more benchmarks compared to NEO (a state-of-the-art synthesis tool), while being 8.71× faster on benchmarks that can be solved by both tools. Furthermore, our ablation study demonstrates the empirical benefits of our proposed reinforcement learning algorithm.

To summarize, this chapter focuses on the following key contributions:

- We propose a new synthesis algorithm based on reinforcement learning that tightly couples statistical and deductive reasoning.

- We describe an off-policy reinforcement learning technique that uses the output of the deduction engine to gradually improve its policy.
- We implement our approach in a tool called CONCORD and empirically demonstrate its benefits compared to other state-of-the-art tools as well as ablations of our own system.

The rest of this chapter is structured as follows. First, we provide some backgound on reinforcement learning and MDPs (Section 3.1) and introduce our problem formulation in Section 3.2. After formulating the synthesis problem as an MDP in Section 3.3, we then present our synthesis algorithm in Section 3.4. Section 3.5 and Section 3.6 describe our implementation and evaluation respectively.

3.1 Background on Reinforcement Learning

At a high level, the goal of reinforcement learning (RL) is to train an agent, such as a robot, to make a sequence of decisions (e.g., move up/down/left/right) in order to accomplish a task. All relevant information about the environment and the task is specified as a *Markov decision process (MDP)*. Given an MDP, the goal is to compute a policy that specifies how the agent should act in each state to maximize their chances of accomplishing the task.

In the remainder of this section, we provide background on MDPs and describe the policy gradient algorithm that our method will build upon.

Markov Decision Process We formalize a *Markov decision process (MDP)* as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of *states* (e.g., the robot's current position),

- $\mathcal{S}_I$ is the initial state distribution,
- $\mathcal{S}_T$ is a set of the final states (e.g., a dead end),
- $\mathcal{A}$ is a set of actions (e.g., move up/down/left/right),
- $\mathcal{F} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a set of transitions,
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is a reward function that assigns a reward to each state (e.g., 1 for reaching the goal and 0 otherwise).

In general, transitions in an MDP can be stochastic; however, for our setting, we only consider deterministic transitions and rewards.

Policy A policy for an MDP specifies how the agent should act in each state. Specifically, we consider a (stochastic) *policy* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\pi(S, A)$ is the probability of taking action A in state S. Alternatively, we can also think of π as a mapping from states to distributions over actions. Thus, we write $A \sim \pi(S)$ to denote that action A is sampled from the distribution for state s.

Rollout Given an MDP $\mathcal{M}$ and policy π, a *rollout* is a sequence of state-action-reward tuples obtained by sampling an initial state and then using π to make decisions until a final state is reached. More formally, for a rollout of the form:

$$\zeta = ((S_1, A_1, R_1), ..., (S_{m-1}, A_{m-1}, R_{m-1}), (S_m, \varnothing, R_m)),$$

we have $S_m \in \mathcal{S}_T$, $S_1 \sim \mathcal{S}_I$ (i.e., S_1 is sampled from an initial state), and, for each $i \in \{1, ..., m-1\}$, $A_i \sim \pi(S_i)$, $R_i = \mathcal{R}(S_i)$, and $S_{i+1} = \mathcal{F}(S_i, A_i)$.

In general, a policy π induces a distribution $\mathcal{D}_\pi$ over the rollouts of an MDP $\mathcal{M}$. Since we assume that MDP transitions are deterministic, we have:

$$\mathcal{D}_\pi(\zeta) = \prod_{i=1}^{m-1} \pi(S_i, A_i).$$

RL Problem Given an MDP $\mathcal{M}$, the goal of reinforcement learning is to compute an *optimal* policy π^* for $\mathcal{M}$. More formally, π^* should maximize *cumulative expected reward*:

$$\pi^* = \arg\max_\pi J(\pi)$$

where the *cumulative expected reward* $J(\pi)$ is computed as follows:

$$J(\pi) = \mathbb{E}_{\zeta \sim \mathcal{D}_\pi}\left[\sum_{i=1}^{m} R_i\right]$$

Policy Gradient Algorithm The *policy gradient algorithm* is a well-known RL algorithm for finding optimal policies. It assumes a parametric policy family π_θ with parameters $\theta \in \mathbb{R}^d$. For example, π_θ may be a deep neural network (DNN), where θ denotes the parameters of the DNN. At a high level, the policy gradient algorithm uses the following theorem to optimize $J(\pi_\theta)$ [102]:

Theorem 3.1.1. We have

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\zeta \sim \mathcal{D}_{\pi_\theta}}[\ell(\zeta)] \quad \text{where} \quad \ell(\zeta) = \sum_{i=1}^{m-1}\left(\sum_{j=i+1}^{m} R_j\right) \nabla_\theta \log \pi_\theta(S_i, A_i). \tag{3.1}$$

In this theorem, the term $\nabla_\theta \log \pi_\theta(S_i, A_i)$ intuitively gives a direction in the parameter space that, when moving the policy parameters towards it, increases the probability of taking action A_i at state S_i. Also, the sum $\sum_{j=i+1}^{m} R_j$ is the total future reward after taking action A_i . Thus, $\ell(\zeta)$ is just the sum of different directions in the parameter

space weighted by their corresponding future reward. Thus, the gradient $\nabla_\theta J(\pi_\theta)$ moves policy parameters in a direction that increases the probability of taking actions that lead to higher rewards.

Based on this theorem, we can estimate the gradient $\nabla_\theta J(\pi_\theta)$ using rollouts sampled from $\mathcal{D}_{\pi_\theta}$:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(\zeta^{(k)}), \tag{3.2}$$

where $\zeta^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$. The policy gradient algorithm uses stochastic gradient ascent in conjunction with Equation (3.2) to maximize $J(\pi_\theta)$ [102].

3.2 Problem Formulation

We focus on the setting of syntax-guided synthesis [4]. Specifically, given a domain-specific language (DSL) L and a specification ϕ, our goal is to find a program in L that satisfies ϕ. In the remainder of this section, we formally define our synthesis problem and clarify our assumptions.

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.

Definition 3.2.1. (Partial Program) A *partial 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*, and we say that P is *complete* if it does not contain any holes.

Given a partial program P containing a hole H, we can fill this hole by replacing

$$
\begin{array}{lcl}
S & \rightarrow & N \mid L \\
N & \rightarrow & 0 \mid \ \ldots \ \mid 10 \mid x_i \\
L & \rightarrow & x_i \mid \texttt{take}(L, N) \mid \texttt{drop}(L, N) \mid \texttt{sort}(L) \\
 & & \mid \texttt{reverse}(L) \mid \texttt{add}(L, L) \mid \texttt{sub}(L, L) \mid \texttt{sumUpTo}(L)
\end{array}
$$

Figure 3.2: A simple programming language used for illustration. Here, `take` (resp. `drop`) keeps (resp. removes) the first N elements in the input list. Also, `add` (resp. `sub`) compute a new list by adding (resp. subtracting) elements from the two lists pair-wise. Finally, `sumUpTo` generates a new list where the i'th element in the output list is the sum of all previous elements (including the i'th element) in the input list.

H with the right-hand-side of any grammar production r of the form $H \rightarrow e$. We use the notation $P \overset{r}{\Rightarrow} P'$ to indicate that P' is the partial program obtained by replacing the first occurrence of H with the right-hand-side of r, and we write $\textsc{Fill}(P, r) = P'$ whenever $P \overset{r}{\Rightarrow} P'$.

Example 3.2.1. Consider the small programming language shown in Figure 3.2 for manipulating lists of integers. The following partial program P over this DSL contains three holes, namely L_1, L_2, N_1:

$$\texttt{add}(L_1, \texttt{take}(L_2, N_1))$$

Now, consider the production $r \equiv L \rightarrow \texttt{reverse}(L)$. In this case, $\textsc{Fill}(P, r)$ yields the following partial program P':

$$\texttt{add}(\texttt{reverse}(L_1), \texttt{take}(L_2, N_1))$$

Program Synthesis Problem Given a specification ϕ and language $L = (V, \Sigma, R, S)$, the goal of program synthesis is to find a *complete* program P such that $S \overset{*}{\Rightarrow} P$ and P satisfies ϕ. We use the notation $P \models \phi$ to indicate that P is a complete program that satisfies specification ϕ.

Deduction Engine In the remainder of this chapter, 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 3.2.2. (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 oracle over-approximates feasibility. That is, if P is feasible with respect to specification ϕ, then DEDUCE(P, ϕ) 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 [109, 45, 58, 47, 43, 47].

Example 3.2.2. Consider again the DSL from Figure 3.2 and the specification ϕ defined by the following input-output example:

$$[65, 2, 73, 62, 78] \mapsto [143, 129, 213, 204, 345]$$

The partial program $\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$ is infeasible because, no matter what production we use to fill non-terminal N, the resulting program cannot satisfy the provided specification for the following reason:

- Given a list l and integer n where $n < \mathit{length}(l)$, $\texttt{take}(l, n)$ returns the first n elements in l. Thus, the length of $\texttt{take}(l, n)$ is smaller than that of l.
- The construct $\texttt{reverse}(l)$ reverses its input; thus, the size of the output list is the same as its input.

- Finally, $\texttt{add}(l_1, l_2)$ constructs a new list by adding the elements of its input lists pair-wise. Thus, `add` expects the two input lists to be the same size.
- Since the outputs of `reverse` and `take` do not have the same size, we cannot combine them using `add`.

Several techniques from prior work (e.g., [43, 45, 109, 85]) can prove the infeasibility of such partial programs by using an SMT solver (provided specifications are given for the DSL constructs).

Beyond checking feasibility, some deduction techniques used for synthesis can also provide additional information [45, 110, 69]. In particular, given a partial program P that is infeasible with respect to specification ϕ, several deduction engines can generate a set of other infeasible partial programs $P_1, \ldots, P_n$ that are infeasible for the same reason as P. To unify both types of feedback, we assume that the output of the deduction oracle $\mathcal{O}$ is a set S of partial programs such that S is empty if and only if $\mathcal{O}$ decides that the partial program is feasible.

This discussion is summarized by the following definition:

Definition 3.2.3. (Deduction Engine) Given a partial program P and specification ϕ, $\textsc{Deduce}(P, \phi)$ yields a set of partial programs S such that (1) if $S \neq \varnothing$, then P is infeasible, and (2) for every $P' \in S$, it must be the case that P' is infeasible with respect to ϕ.

Example 3.2.3. Consider again the same infeasible partial program P given in Example 3.2.2. Since $\texttt{drop}(l, n)$ drops the first n elements from list l (where $n < \mathit{length}(l)$), it also produces a list whose length is smaller than that of the input. Thus, the following partial program P' is also infeasible for the same reason as P:

$$P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$$

Thus, $\textsc{Deduce}(P, \phi)$ may return the set $\{P, P'\}$.

3.3 MDP Formulation of Deduction-Guided Program Synthesis

Given a specification ϕ and language $L = (V, \Sigma, R, S)$, we can formulate the program synthesis problem as an MDP $\mathcal{M}_\phi = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- States $\mathcal{S}$ include all partial programs P such that $S \overset{*}{\Rightarrow} P$ as well as a special label $\perp$ indicating a syntactically ill-formed partial program
- $\mathcal{S}_I$ places all probability mass on the empty program S, i.e.,

$$\mathcal{S}_I(P) = \begin{cases} 1 & \text{if } P = S \\ 0 & \text{if } P \neq S \end{cases}$$

- $\mathcal{S}_T$ includes complete programs as well as infeasible partial programs, i.e.,

$$P \in \mathcal{S}_T \iff \textsc{IsComplete}(P) \ \vee \ \textsc{Deduce}(P, \phi) \neq \varnothing \vee P = \perp$$

- Actions $\mathcal{A}$ are exactly the productions R for the DSL
- Transitions $\mathcal{F}$ correspond to filling a hole using some production i.e.,

$$\mathcal{F}(P,\ r = (H \to e)) = \begin{cases} \perp & \text{if } H \text{ is not a hole in } P \\ \textsc{Fill}(P, r) & \text{otherwise} \end{cases}$$

- The reward function penalizes infeasible programs and rewards correct solutions,

i.e.,

$$\mathcal{R}(P) = \begin{cases} 1 & \text{if } P \models \phi \\ -1 & \text{if } P = \bot \vee \textsc{Deduce}(P, \phi) \neq \varnothing \vee (\textsc{IsComplete}(P) \wedge P \not\models \phi) \\ 0 & \text{otherwise}^1. \end{cases}$$

Observe that our reward function encodes the goal of synthesizing a complete program P that satisfies ϕ, while avoiding the exploration of as many infeasible programs as possible. Thus, if we have a good policy π for this MDP, then a rollout of π is likely to correspond to a solution of the given synthesis problem.

Example 3.3.1. Consider the same specification (i.e., input-output example) ϕ from Example 3.2.2 and the DSL from Example 3.2.1. The partial program

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$$

is a terminal state of $\mathcal{M}_\phi$ since $\textsc{Deduce}(P, \phi)$ yields a non-empty set, and we have $\mathcal{R}(P) = -1$. Thus, the following sequence corresponds to a rollout of $\mathcal{M}_\phi$:

$$(S, S \to L, 0),\ (L, L \to \texttt{add}(L, L), 0),\ (\texttt{add}(L_1, L_2), L \to \texttt{reverse}(L), 0)$$
$$(\texttt{add}(\texttt{reverse}(L_1), L_2), L \to x, 0),\ (\texttt{add}(\texttt{reverse}(x), L), L \to \texttt{take}(L, N), 0)$$
$$(\texttt{add}(\texttt{reverse}(x), \texttt{take}(L, N)), L \to x, 0), (\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)), \varnothing, -1).$$

Simplified Policy Gradient Estimation for $\mathcal{M}_\phi$ Since our synthesis algorithm will be based on policy gradient, we will now derive a simplified policy gradient for our MDP

[1]Note that the rewards are usually tailored and optimized to fit different real-world scenarios. For example, small positive rewards can be assigned to (feasible) intermediate states to stabilize the learning process as well as preventing policy divergence.

$\mathcal{M}_\phi$. First, by construction of $\mathcal{M}_\phi$, a rollout ζ has the form

$$(P_1, r_1, 0), ..., (P_m, \varnothing, q)$$

where $q = 1$ if $P_m \models \phi$ and $q = -1$ otherwise. Thus, the term $\ell(P)$ from Equation (3.1) can be simplified as follows:

$$\ell(P_m) = \sum_{i=1}^{m-1} q \cdot \nabla_\theta \log \pi_\theta(P_i, r_i), \tag{3.3}$$

where $P_m \sim \mathcal{D}_{\pi_\theta}$ is a final state (i.e., complete program or infeasible partial program) sampled using π_θ. Then, Equation (3.1) is equivalently

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(P^{(k)}), \tag{3.4}$$

where $P^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$.

3.4 RL-Based Synthesis Algorithm

In this section, we describe our synthesis algorithm based on reinforcement learning. Our method is an *off-policy* variant of the standard (on-policy) policy gradient algorithm and incorporates additional feedback – in the form of other infeasible programs – provided by the deduction engine when improving its policy parameters. We first give a high-level overview of the synthesis algorithm and then explain how to update the policy.

Algorithm 2 Deduction-Guided Reinforcement Learning for Program Synthesis

1: **procedure** SYNTHESIZE(L, ϕ, π_0)
2: **input:** Domain-specific language $L = (V, \Sigma, R, S)$
3: **input:** Specification ϕ; initial policy π_0
4: **output:** Complete program P such that $P \models \phi$
5: $\pi_\theta \leftarrow \pi_0$
6: **while true do**
7: $(P, \mathcal{C}) \leftarrow$ GETROLLOUT(L, ϕ, π_θ)
8: **if** $\mathcal{C} = \varnothing$ **then return** P
9: **else** $\pi_\theta \leftarrow$ UPDATEPOLICY($\pi_\theta, \mathcal{C}$)

10: **procedure** GETROLLOUT(L, ϕ, π_θ)
11: $P \leftarrow S$
12: **while true do**
13: $\mathcal{C} \leftarrow$ DEDUCE(P, ϕ)
14: **if** $\mathcal{C} \neq \varnothing$ **then return** $(P, \mathcal{C})$
15: **choose** $r \sim \pi_\theta(P) \wedge$ LHS(r) $\in$ HOLES(P)
16: $P \leftarrow$ FILL(P, r)
17: **if** ISCOMPLETE(P) **then**
18: **if** $P \models \phi$ **then return** $(P, \varnothing)$
19: **else return** $(P, \{P\})$

20: **procedure** UPDATEPOLICY($\pi_\theta, \mathcal{C}$)
21: **for** $k' \in \{1, ..., n'\}$ **do**
22: $P^{(k')} \sim \text{Uniform}(\mathcal{C})$
23: $\theta' \leftarrow \theta + \eta \sum_{k'=1}^{n'} \ell(P^{(k')}) \cdot \frac{\mathcal{D}_{\pi_\theta}(P^{(k')})}{1/|\mathcal{C}|}$
24: **return** $\pi_{\theta'}$

3.4.1 Overview of Synthesis Algorithm

Our RL-based synthesis algorithm is presented in Algorithm 2. In addition to specification ϕ and domain-specific language L, this algorithm also takes as input an initial policy π_0 that has been trained off-line on a representative set of training problems[2]. In each iteration of the main synthesis loop, we first obtain a rollout of the current policy by calling the GETROLLOUT procedure at line 7. Here, each rollout either corresponds

[2]We explain how to train this initial policy in Section 3.5.

to a complete program P or an infeasible partial program. If P is complete *and* satisfies the specification, we return it as a solution in line 8. Otherwise, we use feedback $\mathcal{C}$ provided by the deduction engine to improve the current policy (line 9). In the following subsections, we explain the GETROLLOUT and UPDATEPOLICY procedures in more detail.

3.4.2 Sampling Rollouts

The GETROLLOUT procedure iteratively expands a partial program, starting from the start symbol S of the grammar (line 11). In each iteration (lines 12–19), we first check whether the current partial program P is feasible by calling DEDUCE. If P is infeasible (i.e., $\mathcal{C}$ is non-empty), then we have reached a terminal state of the MDP; thus, we return P as the final state of the rollout. Otherwise, we continue expanding P according to the current policy π_θ. Specifically, we first sample an action (i.e., grammar production) r that is applicable to the current state (i.e., the left-hand-side of r is a hole in P), and, then, we expand P by calling the FILL procedure (defined in Section 3.2) at line 16. If the resulting program is complete , we have reached a terminal state and return P; otherwise, we continue expanding P according to the current policy.

3.4.3 Improving the Policy

As mentioned earlier, our algorithm improves the policy by using the feedback $\mathcal{C}$ provided by the deduction engine. Specifically, consider an infeasible program P explored by the synthesis algorithm at line 7. Since DEDUCE(P, ϕ) yields a set of infeasible programs, for every program $P' \in \mathcal{C}$, we know that the reward should be -1. As a consequence, we should be able to incorporate the rollout used to construct P into the policy gradient estimate based on Equation (3.3). However, the challenge to doing so is that Equation (3.4)

relies on *on-policy* samples – i.e., the programs $P^{(k)}$ in Equation (3.4) must be sampled using the current policy π_θ. Since $P' \in \mathcal{C}$ is not sampled using π_θ, we cannot directly use it in Equation (3.4).

Instead, we use *off-policy* RL to incorporate P' into the estimate of $\nabla_\theta J(\pi_\theta)$ [59]. Essentially, the idea is to use *importance weighting* to incorporate data sampled from a different distribution than $\mathcal{D}_{\pi_\theta}$. In particular, suppose we are given a distribution $\tilde{\mathcal{D}}$ over final states. Then, we can derive the following gradient:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{P\sim\mathcal{D}_{\pi_\theta}}[\ell(P)] = \mathbb{E}_{P\sim\tilde{\mathcal{D}}}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}\right].$$

Intuitively, the *importance weight* $\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}$ accounts for the fact that P is sampled from the "wrong" distribution.

Now, we can use the distribution $\tilde{\mathcal{D}} = \text{Uniform}(\textsc{Deduce}(P', \phi))$ for a randomly sampled final state $P' \sim \mathcal{D}_{\pi_\theta}$. Thus, we have: [3]

Theorem 3.4.1. The policy gradient is

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim\text{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right]. \tag{3.5}$$

Proof. Note that

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}}[\nabla_\theta J(\pi_\theta)] \\ &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim\text{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right],\end{aligned}$$

as claimed. □ □

[3] Technically, importance weighting requires that the support of $\tilde{\mathcal{D}}$ contains the support of $\mathcal{D}_{\pi_\theta}$. We can address this issue by combining $\tilde{\mathcal{D}}$ and $\mathcal{D}_{\pi_\theta}$—in particular, take $\tilde{\mathcal{D}}(P) = (1-\epsilon)\cdot \text{Uniform}(\textsc{Deduce}(P',\phi))(P) + \epsilon\cdot\mathcal{D}_{\pi_\theta}(P)$, for any $\epsilon > 0$.

The corresponding estimate of $\nabla_\theta J(\pi_\theta)$ is given by the following equation:

$$\nabla_\theta J(\theta) \approx \frac{1}{n}\sum_{k=1}^{n}\frac{1}{n'}\sum_{k'=1}^{n'}\ell(P^{(k,k')})\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k,k')})}{1/|\textsc{Deduce}(P^{(k)},\phi)|},$$

where $P^{(k)} \sim \mathcal{D}_{\pi_\theta}$ and $P^{(k,k')} \sim \text{Uniform}(\textsc{Deduce}(P^{(k)},\phi))$ for each $k \in \{1,...,n\}$ and $k' \in \{1,...,n'\}$. Our actual implementation uses $n = 1$, in which case this equation can be simplified to the following:

$$\nabla_\theta J(\theta) \approx \frac{1}{n'}\sum_{k'=1}^{n'}\ell(P^{(k')})\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k')})}{1/|\textsc{Deduce}(P,\phi)|}, \tag{3.6}$$

where $P \sim \mathcal{D}_{\pi_\theta}$ and $P^{(k')} \sim \text{Uniform}(\textsc{Deduce}(P,\phi))$ for each $k' \in \{1,...,n'\}$.

Now, going back to our synthesis algorithm from Algorithm 2, the UPDATEPOLICY procedure uses Equation (3.6) to update the policy parameters θ. Specifically, given a set $\mathcal{C}$ of infeasible partial programs, we first sample n' programs $P^{(1)},\ldots,P^{(n')}$ from $\mathcal{C}$ uniformly at random (line 22). Then, we use the probability of each $P^{(k)}$ being sampled from the current distribution $\mathcal{D}_{\pi_\theta}$ to update the policy parameters to a new value θ' according to Equation (3.6).

Example 3.4.1. Suppose that the current policy assigns the following probabilities to these state, action pairs:

$$\pi_\theta((\texttt{add}(\texttt{reverse}(x),L)), L \to \texttt{take}(L,N)) = 0.3,$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x),L)), L \to \texttt{drop}(L,N)) = 0.3,$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x),L)), L \to \texttt{sumUpTo}(L)) = 0.1.$$

Furthermore, suppose that we sample the following rollout using this policy:

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)).$$

This corresponds to an infeasible partial program, and, as in Example 3, DEDUCE(P,ϕ) yields $\{P, P'\}$ where $P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$. Using the gradients derived by Equation (3.6), we update the policy parameters θ to θ'. The updated policy now assigns the following probabilities to the same state, action pairs:

$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{take}(L, N)) = 0.15,$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{drop}(L, N)) = 0.15,$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{sumUpTo}(L)) = 0.2.$$

Observe that the updated policy makes it less likely that we will expand the partial program $\texttt{add}(\texttt{reverse}(x), L))$ using the `drop` production in addition to the `take` production. Thus, if we reach the same state $\texttt{add}(\texttt{reverse}(x), L)$ during rollout sampling in the next iteration, the policy will make it more likely to explore the `sumUpTo` production, which does occur in the desired program

$$\texttt{add}(\texttt{reverse}(x), \texttt{sumUpTo}(x))$$

that meets the specification from Example 2.

3.5 Implementation

We have implemented the proposed algorithm in a new tool called CONCORD written in Python. In what follows, we elaborate on various aspects of our implementation.

3.5.1 Deduction Engine

CONCORD uses the same deduction engine described by NEO [45]. Specifically, given a partial program P, CONCORD first generates a specification φ of P by leveraging the abstract semantics of each DSL construct. Then, CONCORD issues a satisfiability query to the Z3 SMT solver [37] to check whether φ is consistent with the provided specification. If it is not, this means that P is infeasible, and CONCORD proceeds to infer other partial programs that are also infeasible for the same reason as P. To do so, CONCORD first obtains an unsatisfiable core ψ for the queried formula, and, for each clause c_i of ψ originating from DSL construct f_i, it identifies a set S_i of other DSL constructs whose semantics imply c_i. Finally, it generates a set of other infeasible programs by replacing all f_i's in the current program with another construct drawn from its corresponding set S_i.

3.5.2 Policy Network

Architecture As shown by Figure 3.3, CONCORD represents its underlying policy using a deep neural network (DNN) $\pi_\theta(r \mid P)$, which takes as input the current state (i.e., a partial program P) and outputs a probability distribution over actions (i.e., productions r in the DSL). We represent each program P as a flat sequence of statements and use a recurrent neural network (RNN) architecture, as this is a natural choice for sequence inputs. In particular, our policy network is a gated recurrent unit (GRU) network [33], which is a state-of-the-art RNN architecture. Our policy network has one hidden layer with 256 neurons; this layer is sequentially applied to each statement in the partial program together with the latent vector from processing the previous statement. Once the entire partial program P has been encoded into a vector, π_θ has a final layer that outputs a distribution over DSL productions r based on this vector.

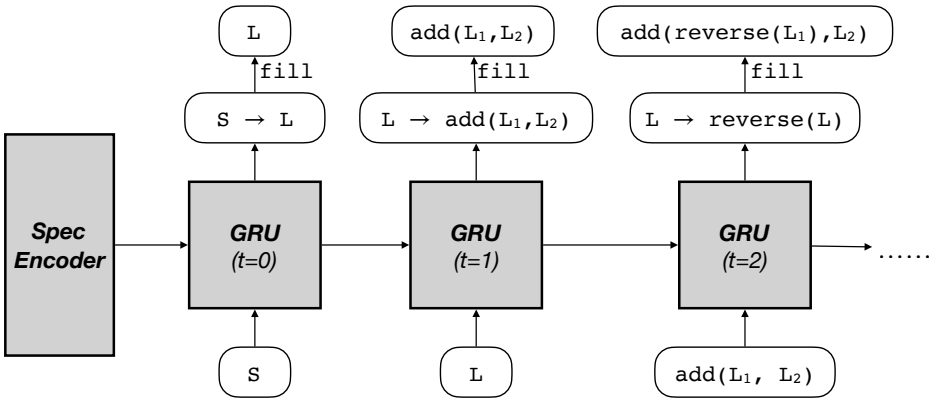


Figure 3.3: The architecture of the policy network showing how to roll out the partial program in Example 4.

Pretraining of Initial Policy Recall from Section 3.4 that our synthesis algorithm takes a input an *initial policy network* that is updated during the synthesis process. One way to initialize the the policy network would be to use a standard random initialization of the network weights. However, a more effective alternative is to *pretrain* the policy on a benchmark suite of program synthesis problems [96]. Specifically, consider a representative training set X_{train} of synthesis problems of the form (ϕ, P), where ϕ is the specification and P is the desired program. To obtain an initial policy, we augment our policy network to take as input an encoding of the specification ϕ for the current synthesis problem – i.e., it has the form $\pi_\theta(r \mid P, \phi)$[4]. Then, we use supervised learning to train π_θ to predict P given ϕ —i.e.,

$$\theta^0 = \arg\max_\theta \sum_{(\phi,P)\in X_{\text{train}}} \sum_{i=1}^{|P|-1} \pi_\theta(r_i \mid P_i, \phi).$$

We optimize θ using stochastic-gradient descent (SGD) on this objective.

Given a new synthesis problem ϕ, we use π_{θ^0} as the initial policy. Our RL algorithm then continues to update the parameters starting from θ^0.

[4]Including the specification as an input to π_θ is unnecessary if we do not use pretraining, since ϕ does not change for a single synthesis problem.

3.5.3 Input Featurization

As standard, we need a way to featurize the inputs to our policy network – i.e., the statements in each partial program P, and the specification ϕ. Our current implementation assumes that statements are drawn from a finite set and featurizes them by training a different embedding vector for each kind of statement. While our general methodology can be applied to different types specifications, our implementation featurizes the specification under the assumption that it consists of input-output examples and uses the same methodology described by Balog et al. [6].

3.5.4 Optimizations

Our implementation performs a few optimization over the algorithm presented in Section 3.4. First, since it is possible to sample the same rollout multiple times, our implementation uses a hash map to check whether a rollout has already been explored. Second, in different invocations of the GETROLLOUT procedure from Algorithm 2, we may end up querying the feasibility of the same state (i.e., partial program) *many* times. Since checking feasibility requires a potentially-expensive call to the SMT solver, our implementation also memorizes the results of feasibility checks for each state. Finally, similar to Chen et al. [25], we use a 3-model ensemble to alleviate some of the randomness in the synthesis process and return a solution as soon as one of the models in the ensemble finds a correct solution.

3.6 Evaluation

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

1. How does CONCORD compare against existing synthesis tools?
2. What is the impact of updating the statistical model during synthesis? (i.e., is reinforcement learning actually useful)?
3. How important is the proposed off-policy RL algorithm compared to standard policy gradient?
4. How important is it to get feedback from the deduction engine when updating the policy?

Benchmarks We evaluate the proposed technique on a total of 100 synthesis tasks used in prior work [6, 45]. Specifically, these synthesis tasks require performing non-trivial transformations and computations over lists using a functional programming language. Since these benchmarks have been used to evaluate both NEO [45] and DEEPCODER [6], they provide a fair ground for comparing our approach against two of the most closely-related techniques. In particular, note that DEEPCODER uses a pre-trained deep neural network to guide its search, whereas NEO uses both statistical and logical reasoning (i.e., statistical model to guide search and deduction to prune the search space). However, unlike our proposed approach, neither NEO nor DEEPCODER update their statistical model during synthesis time.

Training Recall that our algorithm utilizes a pre-trained initial policy. To generate the initial policy, we use the same methodology described in DeepCoder [6] and adopted in NEO [45]. Specifically, we randomly generate both programs and inputs, and we obtain the corresponding output by executing the program. Then, we train the DNN model discussed in Section 3.5 on the Google Cloud Platform with a 2.20GHz Intel Xeon CPU and an NVIDIA Tesla K80 GPU using 16GB of memory.

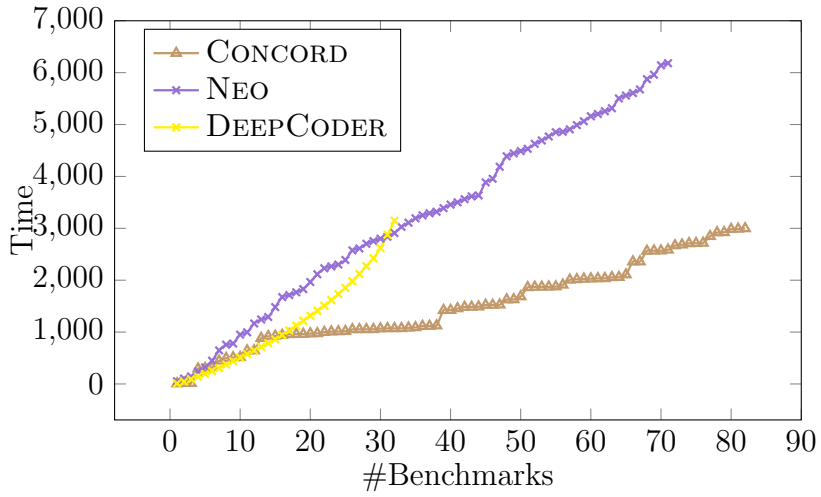


Figure 3.4: Comparison between CONCORD, NEO, and DEEPCODER

3.6.1 Comparison Against Existing Tools

To answer our first research question, we compare CONCORD against both NEO and DeepCoder on the 100 synthesis benchmarks discussed earlier. The result of this comparison is shown in Figure 3.4, which plots the number of benchmarks solved within a given time limit for each of the three tools. As we can see from this figure, CONCORD outperforms DEEPCODER and NEO both in terms of synthesis time as well as the number of benchmarks solved within the 5-minute time limit. In particular, CONCORD can solve 82% of these benchmarks with an average running time of 36 seconds, whereas NEO (resp. DEEPCODER) solves 71% (resp. 32%) with an average running time of 99 seconds (resp. 205 seconds). Thus, we believe these results answer our first research question in a positive way.

3.6.2 Ablation Study

To answer our remaining research questions, we perform an ablation study in which we compare CONCORD against three variants:

- CONCORD**-noRL:** This variant does not use reinforcement learning to update its policy during synthesis. However, it still uses the pre-trained policy to guide search, and it also uses deduction to prune infeasible partial programs. In other words, CONCORD-noRL is the same as the synthesis algorithm from Algorithm 2 but it does not invoke the UPDATEPOLICY procedure to improve its policy during synthesis.

- CONCORD**-NoDeduce:** This variant uses reinforcement learning; however, it does not incorporate feedback from the deduction engine. That is, rather than checking feasibility of partial programs, it instead samples complete programs and uses the percentage of passing input-output examples as the reward signal. Note that this variant of CONCORD essentially corresponds to the technique proposed by Si et al [96].[5]

- CONCORD**-StandardPG:** Recall that our algorithm uses an off-policy variant of the standard policy gradient algorithm to incorporate additional feedback from the deduction engine. To evaluate the benefit of our proposed approach, we created a variant called CONCORD-StandardPG that uses the standard (i.e., on-policy) policy gradient algorithm. In other words, CONCORD-StandardPG implements the same synthesis algorithm from Algorithm 2 except that it uses Theorem 1 to update θ instead of Theorem 2.

[5]We reimplement the RL algorithm proposed in [96] since we cannot directly compare against their tool. Specifically, the policy network in their implementation is tailored to their problem domain.

	# solved	Delta to NEO	Avg. time (s)	Speedup over NEO
CONCORD-noRL	56	-21%	48	1.63×
CONCORD-NoDeduce	65	-8%	21	3.66×
CONCORD-StandardPG	65	-8%	27	2.88×
CONCORD	82	+15%	9	8.71×

Table 3.1: Results of ablation study comparing different variants.

The results from this evaluation are summarized in Table 3.1. Here, the first column labeled "# solved" shows the number of solved benchmarks, and the second column shows percentage improvement over NEO in terms of benchmarks solved. The third column shows average synthesis time for benchmarks that can be solved by *all* variants and NEO. Finally, the last column shows speed-up in terms of synthesis time compared to NEO.

As we can see from this table, all variants are significantly worse than CONCORD in terms of the number of benchmarks that can be solved within a 5-minute time limit [6]. Furthermore, as we can see from the column labeled "Delta to NEO", all of our proposed ideas are important for improving over the state-of-the-art, as NEO outperforms all three variants but not the full CONCORD system, which solves 15% more benchmarks compared to NEO.

Next, looking at the third column of Table 3.1, we see that all three variants of CONCORD are significantly slower compared to CONCORD in terms of synthesis time. While both CONCORD and all of its variants outperform NEO in terms of synthesis time (for benchmarks solved by all tools), CONCORD by far achieves the greatest speed-up over NEO.

In summary, the results from Table 3.1 highlight that all of our proposed ideas (i.e., (1) improving policy at synthesis time; (2) using feedback from deduction; and (3) off-

[6]To understand the improvement brought by the pre-trainedd policy, we also conduct a baseline experiment by using randomly initialized policy in CONCORD. Given the setting, CONCORD can solve as many as 27% of the benchmarks in the given 5-minute time limit.

policy RL) make a significant difference in practice. Thus, we conclude that the ablation study positively answers our last three research questions.

3.7 Summary

In this chapter, we presented a new program synthesis algorithm based on reinforcement learning. Given an initial policy trained off-line, our method uses this policy to guide its search at synthesis time but also gradually improves this policy using feedback obtained from a deductive reasoning engine. Specifically, we formulated program synthesis as a reinforcement learning problem and proposed a new variant of the *policy gradient* algorithm that is better suited to solve this problem. In addition, we implemented the proposed approach in a new tool called CONCORD and evaluated it on 100 synthesis tasks taken from prior work. Our evaluation shows that CONCORD outperforms a state-of-the-art tool by solving 15% more benchmarks with an average speedup of 8.71×. In addition, our ablation study highlights the advantages of our proposed reinforcement learning algorithm.

There are several avenues for future work. First, while our approach is applicable to different DSLs and specifications, our current implementation focuses on input-output examples. Thus, we are interested in extending our implementation to richer types of specifications and evaluating our method in application domains that require such specifications. Another interesting avenue for future work is to integrate our method with other types of deductive reasoning engines. In particular, while our deduction method is based on SMT, it would be interesting to try other methods (e.g., based on types or abstract interpretation) in conjunction with our proposed RL approach.

Chapter 4

POE: Program Synthesis for Neural Prediction Refinement

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 [13] 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 [51] slightly mitigate this challenge through weak supervision without explicitly annotating data with

logical forms, their performance is far from satisfactory [51, 56] due to the quality and quantity of the training data required to infer the hidden logical connections for deriving the answers.

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 4.1. Specifically, POE starts with answers from an off-the-shelf 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 [51]. 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 con-

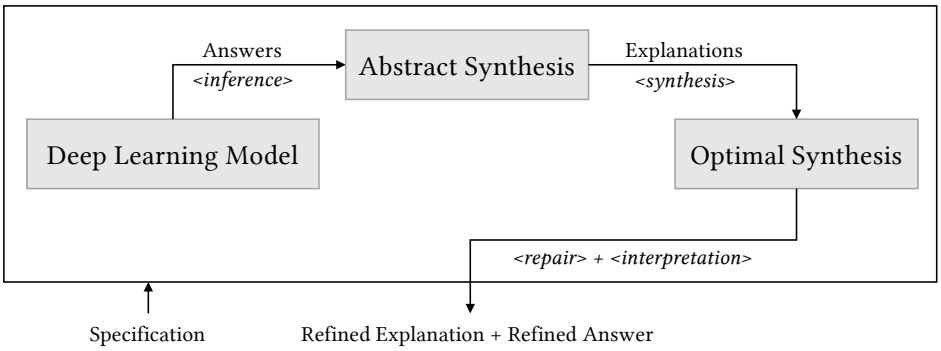


Figure 4.1: Framework overview.

sistent 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 [56], 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 chapter focuses on 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%.

4.1 Overview

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

4.1.1 A Motivating Example

Figure 4.2 (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?"

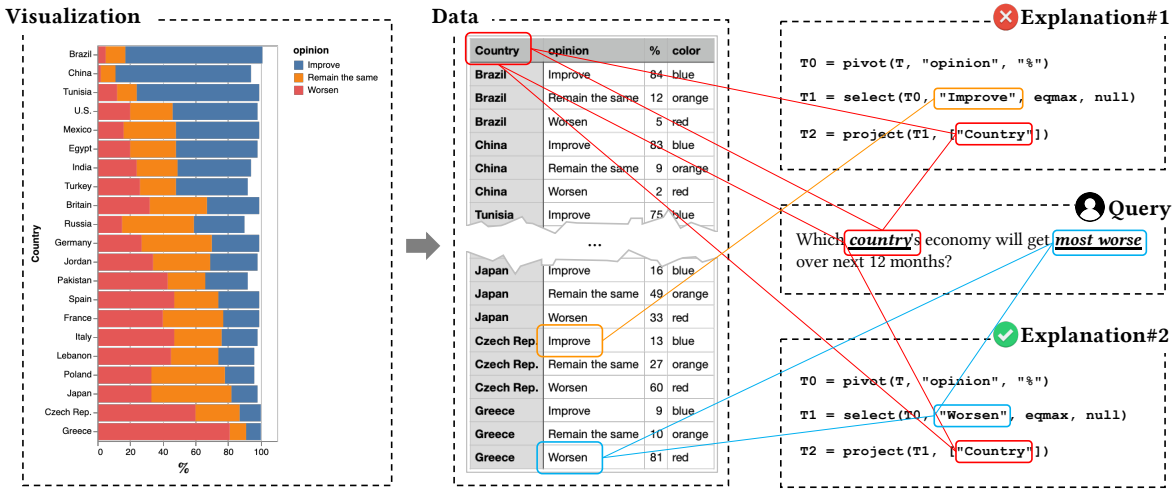


Figure 4.2: A motivating example on data of opinions for future economic growth for different countries. Left: A visualization of stacked bar chart for illustrating the data distribution; Middle: The corresponding table format of the data; Right: Example checking semantic consistency between three parties: 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.

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 [51] 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), ...

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.

4.1.2 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 [56] to obtain a compact representation, as shown in Figure 4.2 (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 4.3, 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

$$
\begin{aligned}
\langle Table \rangle ::= \ & \text{project}(\ \langle Table \rangle, \langle ColList \rangle\ ) \\
| \ & \text{select}(\ \langle Table \rangle, \langle BoolOp \rangle, \langle ColInt \rangle, \langle ConstVal \rangle\ ) \\
| \ & \text{pivot}(\ \langle Table \rangle, \langle ColInt \rangle, \langle ColInt \rangle\ ) \\
| \ & \text{aggregate}(\ \langle Table \rangle, \langle ColList \rangle, \langle AggrOp \rangle, \langle ColInt \rangle\ ) \\
\langle AggrOp \rangle ::= \ & \text{count} \mid \text{min} \mid \text{max} \mid \text{sum} \mid \text{mean} \\
\langle BoolOp \rangle ::= \ & <| \mid <= \mid == \mid >= \mid >| \mid != \mid \text{eqmax} \mid \text{eqmin}
\end{aligned}
$$

$$
\begin{aligned}
\langle Table \rangle \in \textbf{tables}, \ \langle ConstVal \rangle \in \textbf{constants} \\
\langle ColInt \rangle \in \textbf{columns}, \ \langle ColList \rangle \in \textbf{columns}^n
\end{aligned}
$$

Figure 4.3: Syntax of a toy DSL for data wrangling.

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:

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

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.

4.1.3 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 4.2 (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

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

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.

4.2 Preliminaries and Problem Statement

In this section, we first provide some background that will be used throughout the chapter. After that, we describe the architecture of our introspective synthesis algorithm and explain each of its components in detail. However, because both the abstract synthesis and optimal refinement are the main focus of this chapter, we defer a detailed discussion to Section 4.3 and Section 4.4, respectively.

4.2.1 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 4.2.1. Consider the following partial program P:

```
project(◇, ◇)
```

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

```
project(select(◇, ◇, ◇, ◇), ◇)
```

Deduction Engine Motivated by prior work [43, 45, 28] 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 4.2.1. (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 [109, 45, 58, 43, 47].

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

Example 4.2.2. Consider the following input-output example in list manipulation:

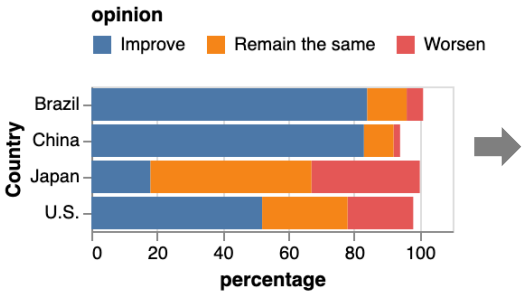
$$e_{in} : [74, 39, 40, 53, 89, 10] \mapsto e_{out} : [78, 80, 106]$$

We use the length of the list as the abstract domain [45]. 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 [51] 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 [56] to convert I into its tabular format. Please refer to Section 4.5 for more details.



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

Figure 4.4: Example tables showing how one can derive similar programs to get conflicting outputs.

4.2.2 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 4.2.3. Figure 4.4 (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"])
```

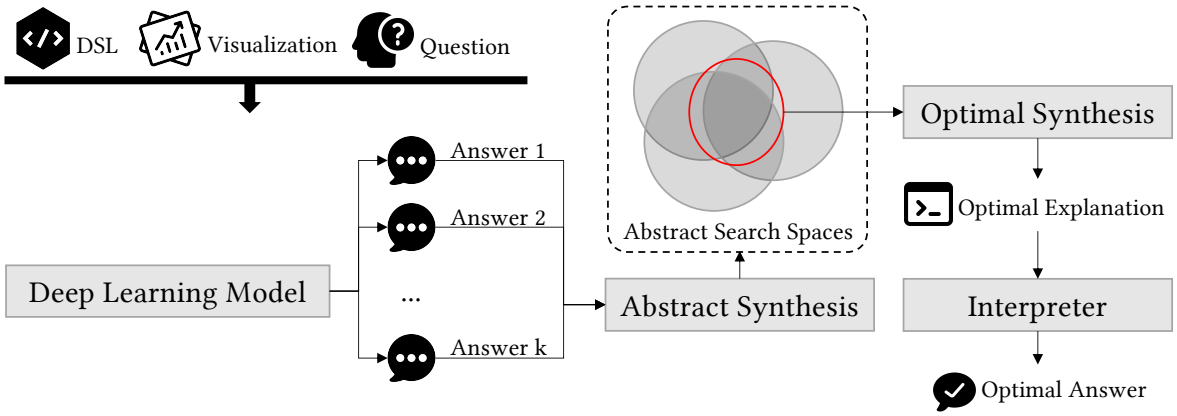


Figure 4.5: System workflow in POE.

where different hole fillings for $\diamond$ will result in different answers, namely "Brazil" (when $\diamond$="Improve") or "Japan" (when $\diamond$="Worsen" or $\diamond$="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 $\mathcal{O}$:

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

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

In particular, we leverage objectives $\mathcal{O}$ to solve a *multi-model* synthesis problem where $\mathcal{O}$ 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 4.2.4. As shown in Figure 4.4, 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 [51]:

"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 4.5 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.

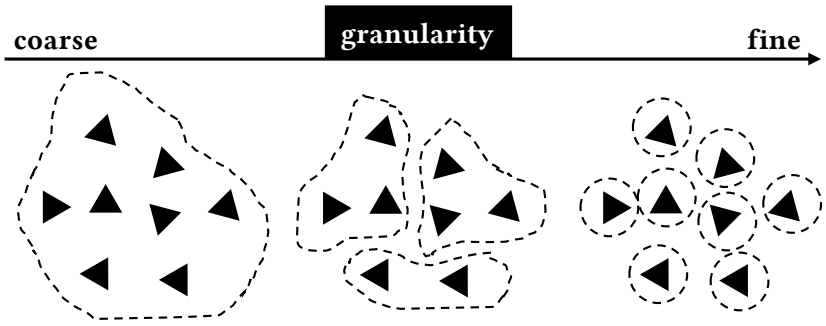


Figure 4.6: Different granularities that affect the algorithm search space. An input-output pair is denoted by a triangle.

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 4.3.

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 4.4.

4.3 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 4.6, given a set of input-output examples E, a naive way (at the left) is to generate a coarse-grained abstract program $\diamond$ 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 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 4.6). 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 4.3.1. (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].

Example 4.3.1. Consider the table shown in Figure 4.4 (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 $\textsc{CountConsist}(P, E)$ 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 3 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 $\perp$, meaning that there is no DSL program that satisfies E.

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

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., $\textsc{CountConsist}(P', E) \leq q$).

Example 4.3.2. Following Example 4.3.1, for the same given IO examples, Algorithm 3 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(◇)
```

Algorithm 3 Abstract Synthesis with Noisy Specification

Input: DSL $\mathcal{L}$, IO Examples E, Balance Coefficient q
Output: Set of Partial Programs $\mathcal{P}$ or $\perp$

1: **procedure** AbsSynth($\mathcal{L}, E, q$)
2: $\mathcal{W} \leftarrow \{\text{Root}(S)\}$
3: **while** true **do**
4: $\mathcal{W}' \leftarrow \mathcal{W}$
5: $\mathcal{W} \leftarrow \mathcal{W} - \{P\}$
6: **for** $P^* \in \{P' \mid \forall d \in \mathcal{L}, P \xrightarrow{d} P'\}$ **do**
7: $n \leftarrow \text{CountConsist}(P^*, E)$
8: **if** $n > q$ **then**
9: $\mathcal{W} \leftarrow \mathcal{W} \cup \{P^*\}$
10: **else if** $n \leq q$ **then**
11: $\mathcal{W} \leftarrow \mathcal{W} \cup \{P\}$
12: **if** $\mathcal{W} = \mathcal{W}'$ **then return** $\mathcal{W}$

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).

4.4 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* [40], which constitutes to one of the major constraints of triangle alignment. A call to the near-synonym linguistic engine $\text{NSYN}(r, s) \in [0, 1]$ 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 4.4.1. Consider the following words: "high", "highest", "low", we have:

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

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 $\mathcal{C}$ 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.

[4]Note that similarity techniqes based on *distributional hypothesis* [50], e.g., word2vec [71] and glove [83], are observably not suitable for distinguishing synonyms and antonyms.

Definition 4.4.1. (0-1 Integer Linear Programming) The 0-1 ILP problem is defined as follows:

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

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 $\mathcal{C}$ 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 $P \in V_P$ generated by Algorithm 3. 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 3.) is used to derive the final solution.

Example 4.4.2. The optimal mapping for Explanation#2 from Figure 4.2 given the following program P:

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

can be represented by the following variables:

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

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 $\mathcal{C}$ 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 $\mathcal{C}$ 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.

$$y_h^t = 0 \text{ if the types of t and h are incompatible.}$$

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 $\mathcal{C}$ 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:

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

Example 4.4.3. Following Figure 4.2 and Example 4.4.2, 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:

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

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:

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

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 4.4.4. Following Example 4.4.3, 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.

4.5 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 [69] framework. In particular, our component specifications are expressed (Section 4.2) 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 [43]. The linguistic engine is built using NLTK (with WordNet interface) [15] and spaCy [73]. 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 [56] and VISER [108], 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 [93] visualization tool to render the visualization from the benchmark specification with extra accessible rich internet application (ARIA) attributes [104], 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 Section 4.3 and Section 4.4. 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 [43, 45], 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 [45], our implementation of COUNTCONSIST performs an additional optimization over Algorithm 3: Since different partial programs may share the same SMT specification, Algorithm 3 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 [3], 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 4.4, 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 [43], 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.

4.6 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?

4. **RQ4 (Ablation):** How significant is the benefit of abstract synthesis (Section 4.3) and optimal alignment (Section 4.4)?

Benchmarks We evaluate POE on a total number of 629 visualization question-answering tasks used in VISQA [56]. 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 [51]. 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 [13], 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].

[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.

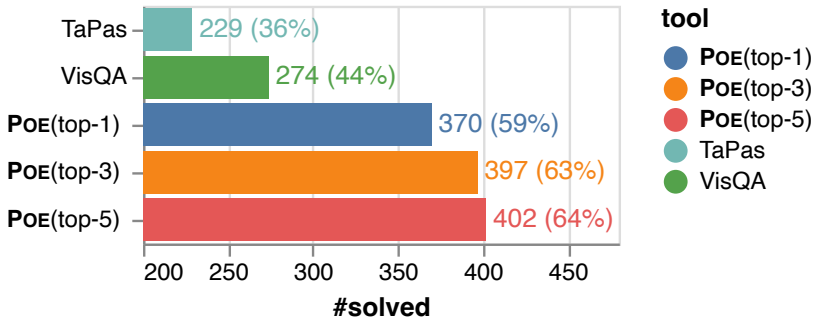


Figure 4.7: Performance comparison between the original pipeline from VISQA (baseline), TAPAS and POE.

4.6.1 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 4.7. 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 4.1. 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.

4.6.2 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$^{\mathcal{O}}$: This variant *only* performs optimal synthesis on the full search space.

Table 4.1: 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%)**

Table 4.2: Comparison between TAPAS and different ablated variants of POE.

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

- 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 4.2 with given timeout of 5 mins. As we can see, without abstract synthesis procedure, POE$^{\mathcal{O}}$ 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.

4.6.3 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 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.

4.6.4 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 [56]. 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.

4.6.5 Discussion

Like any other techniques, our approach also has its own limitations. Based on the result in Figure 4.7, 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 [45, 43]. 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.

4.7 Summary

In this chapter, we 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%.

Chapter 5

Related Work

In this chapter, we discuss prior research that is most related to the addressed topics of this dissertation, including a long line of work on program synthesis, deduction-based reasoning and machine learning, as well as cross-cutting techniques.

5.1 Program Synthesis

Over the past decade, there has been significant interest in automatically synthesizing programs from high-level expressions of user intent [99, 49, 85, 6, 18, 47, 86, 54]. Some of these techniques are geared towards computer end-users and therefore utilize informal specifications such as input-output examples [49, 86, 106], natural language [113, 115, 53, 94], or a combination of both [24, 27]. On the other hand, program synthesis techniques geared towards programmers often utilize additional information, such as a program sketch [99, 103, 42, 76] or types [85, 72] in addition to test cases [44, 65] or logical specifications [103, 18]. While techniques proposed in this dissertation can, in principle, be applied to a broad set of specifications, the particular featurization strategy we use in our implementation is tailored towards input-output examples.

Programming by Example Our techniques are related to a line of work on programming-by-example (PBE) [49, 10, 43, 112, 106, 118]. PBE has been widely applied to different domains such as string manipulation [49, 10], data wrangling [43, 112], and SQL queries [106, 118]. Among these techniques:

- MORPHEUS [43] is directly related to the data wrangling client to which MARS is instantiated. However, unlike MORPHEUS that is specialized to table transformation, the techniques in MARS can be generalized to other synthesis tasks. Compared to existing PBE systems, MARS proposes a novel neural architecture that can learn user preferences from natural language.

- NEO [45] reflects the conflict-driven learning philosophy shared by the core algorithm in CONCORD. While NEO manages the knowledge base following an explicit style of deductive reasoning, CONCORD extends and transplants it to a machine learning model that learns to both prune and propose, thus also reducing extra overheads caused by management. We defer a detailed related work discussion to Section 5.2 and Section 5.3.

- POE's problem setting extends PBE notion by permitting the violation of provided examples, due to the nature of the noisy predictions generated by machine learning models. As a result, POE also shares similar objectives as MARS as they both consider extra layers of specification for determining optimal results.

Programming by Natural Language Programming-by-natural-language (PBNL) is another paradigm [38, 90, 113, 87, 57] that is related to our techniques. Specifically, SQLIZER [113] takes input as natural language and generates its corresponding query in SQL. There are other PBNL systems that translate natural language into simple commands in smartphone [57], IFTTT scripts [87], and scripts for text editing [38, 90].

Compared to previous PBNL systems, our neural architecture can reasonably capture the user intent even in the presence of low quality training data. Furthermore, in addition to natural language, the multi-layer specification in MARS also accepts input-output examples as hard constraints which provide a strong guarantee in correctness. Meanwhile, natural languages also play an important role in POE's encoding of consistency metric between the queries, program constructs and visualizations, as they reflect user intents in synthesis problems.

Interactive Program Synthesis The goal of our techniques also aligns with tools in interactive program synthesis [82, 9, 17], where the goal is to iteratively refine user intent through incorporating user decision in the synthesizer loop. While MARS and POE leverages natural language to capture user intent, we believe the idea of interactive synthesis is complementary to our techniques and can further refine the distribution of the machine learning model.

5.2 Deduction-Based Reasoning

There are many techniques that utilize logical reasoning to perform the core searching and pruning for program synthesis, where the problem is usually formulated a constraint solving instance. Techniques such as conflict-driven learning [14, 117] and counterexample-guided inductive synthesis loop (CEGIS) [4, 60, 98] are closely related to the techniques proposed in this dissertation.

Deduction-Based Pruning The techniques from this dissertation are built upon a line of prior work on using deduction to prune the search space of programs in a DSL [85, 43, 45, 109, 47]. Some of these techniques utilize type-information and type-directed reasoning to detect infeasible partial programs [85, 47, 44, 78, 48]. On the

other hand, other approaches use some form of lightweight program analysis to prune the search space [109, 43, 45]. Concretely, Blaze [109] uses abstract interpretation to build a compact version space representation capturing the space of all feasible programs [109]; Morpheus [43] and Neo [45] utilize logical specifications of DSL constructs to derive specifications of partial programs and query an SMT solver to check for feasibility; Scythe [106] and Viser [108] use deductive reasoning to compute approximate results of partial programs to check their feasibility. Our techniques learn from deduction feedback to improve search efficiency — the deductive reasoning engines used in our implementation for Mars, Concord and Poe are similar to the latter category; however, they can, in principle, be used in conjunction with other deductive reasoning techniques for pruning the search space.

Learning from Failed Synthesis Attempts The techniques proposed in this dissertation can utilize feedback from the deduction engine in the form of other infeasible partial programs. This idea is known as *conflict-driven learning* and has been recently adopted from the SAT solving literature [14, 117] to program synthesis [45]. Specifically, Neo uses the unsat core of the program's specification to derive other infeasible partial programs that share the same root cause of failure, and we use the same idea in our implementation of the deduction engines. While we use logical specifications to infer other infeasible programs, there also exist other techniques (e.g., based on testing [110]) to perform this kind of inference.

5.3 Machine Learning

There has been significant interest in automatically synthesizing programs given high-level specifications of different granularity [99, 49, 85, 6, 18, 47, 86, 54], such as program

sketches [99, 103, 42], types [85, 72], logical forms [103, 18] and natural languages [27, 24, 113, 115, 94]. Recently, machine learning is extensively used for better prioritization of programs for search-based approaches [6, 12, 27, 24, 45]. On the other end, program synthesis techniques and formal methods are also used to provide rich and generalizable feedback for machine learning models [95, 8, 28, 5, 120]. For example, SQLizer [113] performs program repairs based on type-directed program synthesis; Probe [8] utilizes guided bottom-up search to bootstrap machine learning model for synthesis; Metal [96] uses graph-based models of reinforcement learning for synthesis with rewards from SMT solvers.

Machine Learning for Program Synthesis The neural architecture in Mars is relevant to two major directions for applying machine learning to program synthesis. In particular, The first line of work is to directly generate programs from inputs in the form of natural language or input-output examples [74, 75], which is inspired by the seq2seq model in machine translation. Although we also incorporates a seq2seq model as part of the neural architecture, we further leverage the *apriori* algorithm for mining association rules to mitigate the quality of training data.

The second approach [70] incorporates statistical information to guide a program synthesizer. In other words, a statistical model is used to suggest the most promising candidates a synthesizer has to explore. For instance, DeepCoder [6] uses a deep neural network that can directly predict programs from input-output examples. The Morpheus tool [43] adopts an n-gram model for synthesizing data wrangling tasks. Similarly, the SLANG [88] tool integrates an n-gram model for code completion. Raychev et al. [89] extends the previous approach to obtain a statistical model that can guide a synthesizer in the presence of noisy examples. The Neo [45] synthesizer generalizes previous approaches by incorporating an arbitrary statistical model as its "decider" to guide the enumerative

search. While MARS proposes a novel neural architecture to suggest the most promising candidates, it can also leverage advanced techniques from previous work, such as pruning infeasible candidates through deduction [43] and conflict-driven learning [45].

Another other approach from this dissertation, CONCORD, is also related to a long line of work on using machine learning for program synthesis. Among these techniques, some of them train a machine learning model (typically a deep neural network) to directly predict a full program from the given specification [74, 75, 27, 39]. Many of these approaches are based on sequence-to-sequence models [101], sequence to tree models [115], or graph neural networks [94] commonly used in machine translation.

A different approach, sometimes referred to as *learning to search*, is to train a statistical model that is used to *guide* the search rather than directly predict the target program. For example, DEEPCODER [6] uses a deep neural network (DNN) to predict the most promising grammar productions to use for the given input-output examples. Similarly, R3NN [80] and NGDS [55] use DNNs to predict the most promising grammar productions conditioned on both the specification and the current partial program. In addition, there has been work on using concrete program executions on the given input-output examples to guide the DNN [25, 107]. Our technique for pretraining the initial policy network is based on the same ideas as these supervised learning approaches; however, their initial policies do not change during the synthesis algorithm, whereas we continue to update the policy using RL.

While most of the work at the intersection of synthesis and machine learning uses *supervised learning* techniques, recent work has also proposed using reinforcement learning to speed up syntax-guided synthesis [21, 96, 66, 62]. These approaches are all on-policy and do not incorporate feedback from a deduction engine. In contrast, in our problem domain, rewards are very sparse in the program space, which makes exploration highly challenging in a on-policy learning setting. CONCORD addresses this problem using off-

policy RL to incorporate feedback from the deduction engine. Our ablation study results demonstrate that our off-policy RL is able to scale to more complex benchmarks.

Finally, different from prior work [113, 56] that rely on a semantic parser whose training data is difficult and expensive to obtain, the other approach from this dissertation, POE, 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.

Reinforcement Learning for Formal Methods There has been recent interest in applying reinforcement learning (RL) to solve challenging PL problems where large amounts of labeled training data are too expensive to obtain. For instance, Si et al. use graph-based RL to automatically infer loop invariants [95], Singh et al. use Q-learning (a different RL algorithm) to speed up program analysis based on abstract interpretation [97], Dai et al [35] uses meta-reinforcement learning for test data generation, and Chen et al. [22] uses RL to speed up relational program verification. However, these approaches only use RL offline to pretrain a DNN policy used to guide search. In contrast, CONCORD performs reinforcement learning online during synthesis. Bastani et al. has used an RL algorithm called Monte-carlo tree search (MCTS) to guide a specification inference algorithm [11]; however, their setting does not involve any kind of deduction.

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. [91] 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. [34] analyzed the attention mechanisms of pre-trained models and

demonstrated syntactic information captured in these models. Petroni et al. [84] 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 [119] and Spider [116]. Their leaderboards have been frequently updated by newly developed encoder and decoder architectures. For example, RAT-SQL [105] 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 [114]) and bottom-up parsers (e.g., SMBOP [92]) 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 [51] that do not rely on annotating logic forms, can be trained on larger corpora, thus outperform state-of-the-arts. Our evaluation of Poe shows that our introspective synthesis approach that reconciles the power of symbolic reasoning and machine learning can significantly push the boundary of visualization queries.

Chapter 6

Conclusion

This dissertation proposes a synergistic framework that bridges statistical and logical reasoning for program synthesis. We presents the key insights of the framework via three aspects: the interface, the core and the extent. The interface, encodes user-provided specification of multi-modalities into machine-readable constraints, via a hybrid design that exploits the power of both statistical and logical reasoning. The core, resides with a synergistic solution that tightly couples statistical and logical reasoning in program synthesis, where logical feedback is seamlessly incorporated into the search of statistical learning via the new paradigm of deduction-guided reinforcement learning. The extent, shows the potential of the framework by connecting the interface and core with broader interdisciplinary scenarios, where we demonstrates its power by refinement of deep learning model's predictions via program synthesis. We have implemented the framework in three tools, namely Mars, Concord and Poe, and we show in evaluations that they are effective for the core task of program synthesis, as well as in improving experience for end-user programming via broaden expressiveness, enhanced explainability and natural interactivity.

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93*, New York, New York, USA, 1993. ACM Press.

[3] Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. In Ruzica Piskac and Rayna Dimitrova, editors, *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, volume 229 of *EPTCS*, pages 67–83, 2016.

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M K Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, October 2013.

[5] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 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, pages 731–744, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *Proceedings of ICLR'17*, March 2017.

[7] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[8] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA):1–29, November 2020.

[9] Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 121–136, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 218–228, New York, NY, USA, June 2015. Association for Computing Machinery.

[11] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 678–692, New York, NY, USA, June 2018. Association for Computing Machinery.

[12] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA):1–27, October 2019.

[13] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from Question-Answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2008.

[15] Steven Bird and Edward Loper. NLTK: The natural language toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 214–217, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[16] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - an optimizing SMT solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[17] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism.

In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 339–352, New York, NY, USA, January 2010. Association for Computing Machinery.

[18] James Bornholt and Emina 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, pages 467–481, New York, NY, USA, June 2017. Association for Computing Machinery.

[19] P Bose, D Das, Y Chen, Y Feng, C Kruegel, and G Vigna. SAILFISH: Vetting smart contract State-Inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1235–1252, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

[20] Marc Brockschmidt, Miltos Allamanis, Alex Gaunt, and Alex Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, May 2019.

[21] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[22] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. Relational verification using reinforcement learning. *Proc. ACM Program. Lang.*, 3(OOPSLA):1–30, October 2019.

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[24] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 487–502, New York, NY, USA, June 2020. Association for Computing Machinery.

[25] Xinyun Chen, Chang Liu, and Dawn Song. Execution-Guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[26] Yanju Chen and Rong Pan. Automatic emphatic information extraction from aligned acoustic data and its application on sentence compression. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 3422–3428, San Francisco, California, USA, 2017. AAAI Press.

[27] Yanju Chen, Ruben Martins, and Yu Feng. Maximal Multi-Layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 602–612, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using Deduction-Guided reinforcement learning. In Shuvendu K Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 587–610, Cham, 2020. Springer International Publishing.

[29] Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. Tree traversal synthesis using Domain-Specific symbolic compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1030–1042, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Işil Dillig. Synthesis-Powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022.

[31] Yanju Chen, Xifeng Yan, and Yu Feng. Visualization question answering using introspective program synthesis. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 137–151, New York, NY, USA, 2022. Association for Computing Machinery.

[32] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. Fast and reliable program synthesis via user interaction. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, New York, NY, USA, 2024. Association for Computing Machinery.

[33] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN Encoder–Decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*

(EMNLP), pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[34] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy, August 2019. Association for Computational Linguistics.

[35] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[36] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, August 2003.

[37] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[38] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, New York, NY, USA, May 2016. Association for Computing Machinery.

[39] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-Rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017.

[40] Philip Edmonds and Graeme Hirst. Near-synonymy and lexical choice. *Comput. Linguist. Assoc. Comput. Linguist.*, 28(2):105–144, June 2002.

[41] efficient-apriori. efficient-apriori 0.4.5. `https://pypi.org/project/efficient-apriori/`, 2018.

[42] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from Hand-Drawn images. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 6062–6071, Red Hook, NY, USA, 2018. Curran Associates Inc.

[43] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-Based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 422–436, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 599–612, New York, NY, USA, January 2017. Association for Computing Machinery.

[45] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using Conflict-Driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 420–435, New York, NY, USA, 2018. Association for Computing Machinery.

[46] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Morpheus. `https://utopia-group.github.io/morpheus/`, 2018.

[47] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 229–239, New York, NY, USA, June 2015. Association for Computing Machinery.

[48] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, January 2016. Association for Computing Machinery.

[49] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330, New York, NY, USA, January 2011. Association for Computing Machinery.

[50] Zellig S Harris. Distributional structure. *Word World*, 10(2-3):146–162, August 1954.

[51] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. TaPas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333, Online, July 2020. Association for Computational Linguistics.

[52] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, November 1997.

[53] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA, 2018. Association for Computational Linguistics.

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

[55] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-Guided deductive search for Real-Time program synthesis from examples. In *International Conference on Learning Representations*, 2018.

[56] Dae Hyun Kim, Enamul Hoque, and Maneesh Agrawala. Answering questions about charts and generating visual explanations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, New York, NY, USA, April 2020. Association for Computing Machinery.

[57] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth: synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, MobiSys '13, pages 193–206, New York, NY, USA, June 2013. Association for Computing Machinery.

[58] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pages 70–80, New York, NY, USA, October 2016. Association for Computing Machinery.

[59] Sergey Levine and Vladlen Koltun. Guided policy search. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1–9, Atlanta, Georgia, USA, 2013. PMLR.

[60] A Solar Lezama. *Program synthesis by sketching*. PhD thesis, Citeseer, 2008.

[61] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy,

Daniel J Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022.

[62] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 10015–10027, Red Hook, NY, USA, 2018. Curran Associates Inc.

[63] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. Conflict-Driven synthesis for layout engines. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[64] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, August 2017. Association for Computing Machinery.

[66] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The Neuro-Symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019.

[67] B Mariano, Y Chen, Y Feng, S K Lahiri, and I Dillig. Demystifying loops in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–274, Los Alamitos, CA, USA, September 2020. IEEE Computer Society.

[68] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using Neural-Guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[69] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *Proceedings VLDB Endowment*, 12(12):1914–1917, August 2019.

[70] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on*

Machine Learning, volume 28 of *Proceedings of Machine Learning Research*, pages 187–195, Atlanta, Georgia, USA, 2013. PMLR.

[71] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C J Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[72] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP):1–28, July 2019.

[73] 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. explosion/spaCy: v3.2.0: Registered scoring functions, Doc input, floret vectors and more, November 2021.

[74] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*, 2016.

[75] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. In *International Conference on Learning Representations*, 2017.

[76] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR, 2019.

[77] Jetbrains s r o. IntelliJ IDEA. `https://www.jetbrains.com/idea/`. Accessed: 2023-NA-NA.

[78] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, June 2015. Association for Computing Machinery.

[79] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of Under-Constrained circuits in Zero-Knowledge proofs. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[80] Emilio Parisotto, Abdel-Rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

[81] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, High-Performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[82] Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1114–1124, New York, NY, USA, May 2018. Association for Computing Machinery.

[83] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[84] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 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)*, pages 2463–2473, Hong Kong, China, November 2019. Association for Computational Linguistics.

[85] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, June 2016. Association for Computing Machinery.

[86] Oleksandr Polozov and Sumit 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, pages 107–126, New York, NY, USA, 2015. Association for Computing Machinery.

[87] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for If-This-Then-That recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China, July 2015. Association for Computational Linguistics.

[88] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, June 2014. Association for Computing Machinery.

[89] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, New York, NY, USA, January 2016. Association for Computing Machinery.

[90] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 792–800, Buenos Aires, Argentina, 2015. AAAI Press.

[91] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "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*, KDD '16, pages 1135–1144, New York, NY, USA, August 2016. Association for Computing Machinery.

[92] Ohad Rubin and Jonathan Berant. SmBoP: Semi-autoregressive bottom-up semantic parsing. In *Proceedings of the 5th Workshop on Structured Prediction for NLP (SPNLP 2021)*, pages 12–21, Online, August 2021. Association for Computational Linguistics.

[93] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, January 2017.

[94] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 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, 2019.

[95] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Proceedings of the 32nd International*

Conference on Neural Information Processing Systems, NIPS'18, pages 7762–7773, Red Hook, NY, USA, 2018. Curran Associates Inc.

[96] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a Meta-Solver for Syntax-Guided program synthesis. In *International Conference on Learning Representations*, 2019.

[97] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast numerical program analysis with reinforcement learning. In *Computer Aided Verification*, pages 211–229. Springer International Publishing, 2018.

[98] Armando Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems*, pages 4–13. Springer Berlin Heidelberg, 2009.

[99] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 404–415, New York, NY, USA, October 2006. Association for Computing Machinery.

[100] Stackoverflow. Stackoverflow. `https://stackoverflow.com/`, 2018.

[101] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[102] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[103] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for Solver-Aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. Association for Computing Machinery.

[104] W3C. Accessible rich internet applications (WAI-ARIA) 1.1. `https://www.w3.org/TR/wai-aria/`, 2017. Accessed: 2021-11-14.

[105] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 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*, pages 7567–7578, Online, July 2020. Association for Computational Linguistics.

[106] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from Input-Output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. Association for Computing Machinery.

[107] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust Text-to-SQL generation with Execution-Guided decoding, 2018.

[108] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. Visualization by example. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[109] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL):1–30, December 2017.

[110] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 286–300, New York, NY, USA, June 2019. Association for Computing Machinery.

[111] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. In *33rd USENIX Security Symposium (USENIX Security 24)*, PHILADELPHIA, PA, August 2024. USENIX Association.

[112] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 508–521, New York, NY, USA, June 2016. Association for Computing Machinery.

[113] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA): 1–26, October 2017.

[114] Pengcheng Yin and Graham Neubig. A syntactic neural model for General-Purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[115] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. SyntaxSQLNet: Syntax tree networks for complex and Cross-Domain Text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663, Brussels, Belgium, 2018. Association for Computational Linguistics.

[116] 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. 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*, pages 3911–3921, Brussels, Belgium, 2018. Association for Computational Linguistics.

[117] Lintao Zhang, C F Madigan, M H Moskewicz, and S Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 279–285, November 2001.

[118] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234, November 2013.

[119] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning, 2017.

[120] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 686–701, New York, NY, USA, June 2019. Association for Computing Machinery.