UNIVERSITY OF CALIFORNIA SAN DIEGO

Neuro-Symbolic Program Synthesis for Data-Efficient Learning

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

in

Computer Science

by

Shraddha Govind Barke

Committee in charge:

Professor Nadia Polikarpova, Chair
Professor Eric Baković
Professor Taylor Berg-Kirkpatrick
Professor Sorin Lerner

2024

The Dissertation of Shraddha Govind Barke is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

For Baba, who was my pillar of strength, for Aai, whose love inspires me everyday and for Dada, who always has my back.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am incredibly grateful to Nadia Polikarpova for her constant support throughout my PhD journey. Nadia taught me to think critically, to identify and tackle research problems, to present findings through writing and talks in a compelling manner, and to collaborate effectively. Her advice and patience were crucial in making this thesis possible.

I am thankful to my committee members and collaborators at UCSD for their enthusiasm, support, and insights: Taylor Berg-Kirkpatrick, who taught me how to present interdisciplinary research to the machine learning community; Eric Baković, who introduced me to computational linguists and advised me on the phonological rule inference project; Hila Peleg, who collaborated with me on guided program synthesis in the middle of a pandemic; and my amazing lab mates, with whom I had the good fortune to co-author several papers.

I am thankful to Julia Lawall for introducing me to the world of research, helping me with grad school applications, and for her steadfast belief in me from day one. During my Ph.D., my two amazing internships at Microsoft Research gave me invaluable insights into how research can be practically beneficial to users. I would like to thank all my collaborators and mentors at Microsoft who have helped me over the course of my graduate education: Andrew Gordon, Sumit Gulwani, Alan Leung, Carina Negreanu, Christian Poelitz, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, Jack Williams, and Benjamin Zorn.

Last but very importantly, I am thankful to my friends and family across the world who have provided unwavering support and camaraderie, enriching my PhD years.

Chapter 2, in full, is a reprint of the material as it appears in Constraint-based Learning of Phonological Processes. Barke, Shraddha; Kunkel, Rose; Polikarpova, Nadia; Meinhardt, Eric; Baković, Eric; and Bergen, Leon. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. EMNLP-IJCNLP 2019. The dissertation author was a primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in Just-in-Time Learning

for Bottom-Up Enumerative Synthesis. Barke, Shraddha; Peleg, Hila; and Polikarpova, Nadia. Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA. 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in Grounded Copilot: How Programmers Interact with Code-Generating Models. Barke, Shraddha; James, Michael B.; and Polikarpova, Nadia. Proceedings of the ACM on Programming Languages, Volume 7, Issue OOPSLA1. The dissertation author was a primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in Solving Data-centric Tasks using Large Language Models. Barke, Shraddha; Poelitz, Christian; Negreanu, Carina; Zorn, Benjamin; Cambronero, José; Gordon, Andrew; Le, Vu; Nouri, Elnaz; Polikarpova, Nadia; Sarkar, Advait; Slininger, Brian; Toronto, Neil; Williams, Jack. Findings of the Association for Computational Linguistics: NAACL 2024. The dissertation author was a primary investigator and author of this paper.

Chapter 6, in full, has been submitted for publication of the material as it may appear in HySynth: Context-Free LLM Approximation for Guiding Program Synthesis. Barke, Shraddha; Gonzalez, Emmanuel; Kasibatla, Saketh; Berg-Kirkpatrick, Taylor; and Polikarpova, Nadia. The dissertation author was the primary investigator and author of this paper.

VITA

2017	Bachelor of Engineering, Electronics and Instrumentation. Birla Institute of Technology and Science (BITS) Pilani, Goa
2020	Master of Science, Computer Science. University of California San Diego
2024	Doctor of Philosophy, Computer Science. University of California San Diego

PUBLICATIONS

Shraddha Barke, Emmanuel Anaya-Gonzalez, Saketh Kasibatla, Taylor Berg-Kirkpatrick, Nadia Polikarpova. "HySynth: Context-Free LLM Approximation for Guiding Program Synthesis." Under Submission to NeurIPS 2024.

Shraddha Barke, Christian Poelitz, Carina Suzana Negreanu, Benjamin Zorn, José Cambronero, Andrew D Gordon, Vu Le, Elnaz Nouri, Nadia Polikarpova, Advait Sarkar, Brian Slininger, Neil Toronto, Jack Williams. "Solving Data-centric Tasks using Large Language Models." Findings of the Association for Computational Linguistics (NAACL). June 2024.

Shraddha Barke, Michael B. James, Nadia Polikarpova. "Grounded Copilot: How Programmers Interact with Code-Generating Models." Proceedings of the ACM on Programming Languages. Volume 7 (OOPSLA). November 2023.

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. "LooPy: Interactive Program Synthesis with Control Structures." Proceedings of the ACM on Programming Languages. Volume 5 (OOPSLA). October 2021.

Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, Ashish Tiwari. "Feedback-Driven Semi-Supervised Synthesis of Program Transformations." Proceedings of the ACM on Programming Languages. Volume 4 (OOPSLA). November 2020.

Shraddha Barke, Hila Peleg, Nadia Polikarpova. "Just-in-time learning for bottom-up enumerative synthesis." Proceedings of the ACM on Programming Languages. Volume 4 (OOPSLA). November 2020.

Shraddha Barke, Rose Kunkel, Nadia Polikarpova, Eric Meinhardt, Eric Baković, Leon Bergen, "Constraint-based Learning of Phonological Processes." Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. EMNLP-IJCNLP 2019.

ABSTRACT OF THE DISSERTATION

Neuro-Symbolic Program Synthesis for Data-Efficient Learning

by

Shraddha Govind Barke

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Nadia Polikarpova, Chair

The dream of intelligent assistants to enhance programmer productivity has now become a concrete reality, with rapid advances in artificial intelligence. Large language models (LLMs) have demonstrated impressive capabilities in various domains based on the vast amount of data used to train them. However, tasks which require structured reasoning or those underrepresented in their training data continue to be a challenge for LLMs.

Program synthesis offers an alternative approach to learning, particularly effective in data-efficient domains with limited training data. It focuses on searching for a program in a domain-specific language that satisfies a given user intent. Program synthesis enables learning of interpretable models that provide correctness and generalizability guarantees from a few

data points leading to data-efficient learning. However, purely symbolic methods based on combinatorial search scale poorly to complex problems. To address these challenges, a hybrid paradigm called neurosymbolic synthesis is being explored. This approach integrates the best of both worlds by combining neural networks with symbolic reasoning, thereby enhancing the robustness of AI assistants.

This dissertation includes technical contributions spanning symbolic, neurosymbolic and neural approaches to program synthesis. It explores the application of *symbolic* constraint-based synthesis in SYPHON to model human language, *hybrid* techniques in PROBE and HYSYNTH that guide symbolic search with a probabilistic model, and *neural* LLM-driven code generation to automate spreadsheet tasks for end users. Additionally, it focuses on strategies to improve user experience by developing more intuitive and user-friendly programming assistants for the future.

Chapter 1

Introduction

Program synthesis is the task of searching for a program in the underlying domain-specific language (DSL) that satisfies a given user intent. The program synthesis problem is parametrized by user specification format, program space and search strategy. The synthesizer allows a user to express their intent usually in the form of input-output examples [5, 74] or natural language [34, 181]. It also takes as input a program space usually defined by a DSL that constrains the space of possible programs. It then searches for a program that satisfies the specification using the structural constraints and returns it to the user. Program synthesis shines in data-efficient domains with restrictive DSLs, generating correct and interpretable models of code. It has been successfully applied to a variety of problem domains, such as string manipulation [74], data wrangling [45], program transformation [67], graphics [50, 84], program repair [100], and superoptimization [142]. However, the combinatorial search space explosion of synthesis has been an obstacle to the wide applicability of these techniques to real-world domains.

More recently, breakthroughs in LLMs have paved the way for powerful programming assistants like Github Copilot [64] which can generate code in a few seconds based on natural language. Despite their remarkable success in complex tasks across various domains, LLMs struggle with structured reasoning and domain-specific tasks that are not adequately represented in their training data. Fine-tuning and prompting techniques also require high-quality data, which is hard to obtain, as these models are sensitive to the training data quality.

Scientists have worked for decades to advance automated programming since the dawn of symbolic AI [113] to the present era, marked by the advent of LLMs [16]. This thesis makes three kinds of contributions towards that goal: (1) Expands the scope of program synthesis applications by illustrating how constraint-based synthesis can be effectively applied outside of traditional programming settings, like modeling human language; (2) Develops hybrid synthesis techniques that guide symbolic search towards more likely programs using a model. This tackles the combinatorial explosion challenge of symbolic techniques by search space prioritization and the data-efficiency challenge of neural techniques by integrating them with structured reasoning; and (3) Explores LLM-driven code generation to automate end user programming in spreadsheets and provides insights to develop intuitive and user-friendly programming assistants for the future.

Chapter 2: Constraint-based Program Synthesis of Phonological Processes

Phonological processes are context-dependent sound changes in natural languages. This chapter presents an unsupervised approach to learning human-readable descriptions of phonological processes from collections of related utterances. Our approach builds upon a technique from the programming languages community called *constraint-based program synthesis* [150]. We contribute a novel encoding of the learning problem into Boolean Satisfiability constraints, which enables both data efficiency and fast inference. We evaluate our system on textbook phonology problems and datasets from the literature, and show that it achieves high accuracy at interactive speeds.

Chapter 3: Just-in-Time Learning for Bottom-Up Enumerative Synthesis

A key challenge in program synthesis is the astronomical size of the search space the synthesizer has to explore. In response to this challenge, prior work proposed to guide synthesis using learned probabilistic models [102]. Obtaining such a model, however, might be infeasible for a problem domain where no high-quality training data is available. In this chapter, we introduce an alternative approach to guided program synthesis: instead of training a model ahead of time we show how to bootstrap one just in time, during synthesis, by learning from partial

solutions encountered along the way. To make the best use of the model, we also propose a new program enumeration algorithm we dub guided bottom-up search, which extends the efficient bottom-up search with guidance from probabilistic models. We implement this approach in a tool called PROBE, which targets problems in the popular syntax-guided synthesis (SyGuS) format. We evaluate PROBE on benchmarks from the literature and show that it achieves significant performance gains both over unguided bottom-up search and over a state-of-the-art probability-guided synthesizer, which had been trained on a corpus of existing solutions.

Chapter 4: Grounded Copilot: How Programmers Interact with Code-Generating Models

Powered by recent advances in code-generating models, AI assistants like Github Copilot promise to change the face of programming forever. This chapter presents a grounded theory analysis of how programmers interact with Copilot, based on observing 20 participants—with a range of prior experience using the assistant—as they solve diverse programming tasks across four languages. Our main finding is that interactions with programming assistants are bimodal: in acceleration mode, the programmer knows what to do next and uses Copilot to get there faster; in exploration mode, the programmer is unsure how to proceed and uses Copilot to explore their options. Based on our theory, we provide recommendations for improving the usability of future AI programming assistants.

Chapter 5: Solving Data-centric Tasks using Large Language Models

Large language models (LLMs) are rapidly replacing help forums like StackOverflow, and are especially helpful for non-professional programmers and end users. These users are often interested in data-centric tasks, such as spreadsheet manipulation and data wrangling, which are hard to solve if the intent is only communicated using a natural-language description, without including the data. But how do we decide how much data and which data to include in the prompt? This chapter discusses two contributions we make towards answering this question. First, we create a dataset of real-world NL-to-code tasks manipulating tabular data, mined from StackOverflow posts. Second, we introduce a cluster-then-select prompting technique, which

adds the most representative rows from the input data to the LLM prompt. Our experiments show that LLM performance is indeed sensitive to the amount of data passed in the prompt, and that for tasks with a lot of syntactic variation in the input table, our cluster-then-select technique outperforms a random selection baseline.

Chapter 6: HySynth: Context-Free LLM Approximation for Guiding Program Synthesis

Many structured prediction and reasoning tasks can be framed as program synthesis problems, where the goal is to generate a program in a domain-specific language (DSL) that transforms input data into the desired output. Unfortunately, purely neural approaches, such as large language models (LLMs), often fail to produce fully correct programs in unfamiliar DSLs, while purely symbolic methods based on combinatorial search scale poorly to complex problems. Motivated by these limitations, this chapter introduces a hybrid neurosymbolic approach, where LLM completions for a given task are used to learn a task-specific, context-free surrogate model, which is then used to guide program synthesis. We evaluate this hybrid approach on three domains, and show that it outperforms both unguided search and direct sampling from LLMs, as well as existing program synthesizers.

Chapter 2

Constraint-based Program Synthesis of Phonological Processes

2.1 Introduction

Phonological processes govern the way speech sounds in natural languages change depending on the context. For example, in English verbs, the past tense suffix /d/ turns into [t] after voiceless consonants (so the word "zipped" is pronounced [zɪp**t**], while "begged" is pronounced [bɛg**d**]). Linguists routinely face the task of inferring phonological processes by observing and contrasting *surface forms* (pronunciations) of morphologically related words. To aid linguists with this task, we consider the problem of learning phonological processes automatically from collections of related surface forms.

This problem setting imposes four core requirements, which guide the design of our approach:

1. Inference results must be **fully interpretable**: our goal is to *explain* phonological processes exhibited by the data, not merely *predict* pronunciations of unseen words. Hence, our model takes the form of discrete, conditional rewrite rules from rule-based phonology [37].

2. Inference must be **unsupervised**: phonological processes are formalized as transformations from (latent) *underlying forms* to surface forms (rather than *between* surface forms).

3. Inference must be **data-efficient**: typically only a handful of data points are available.

4. Inference must be **fast**: we envision linguists using our system interactively, tweaking the data and being able to see the inferred rules within minutes.

Recently *program synthesis* has emerged as a promising approach to interpretable and data-efficient learning [52, 148, 163, 51]. In program synthesis, models are represented as programs in a domain-specific language (DSL), which allows domain experts to impose a strong prior by designing an appropriate DSL. Program synthesis uses powerful constraint solvers to perform combinatorial optimization and find the least-cost program in the DSL that fits the data. Program synthesis has been previously used to tackle the problem of phonological rule learning [52], however their work uses global inference which scales poorly and hence does not satisfy requirement 4 (their system takes an hour on average to solve a phonology textbook problem).

In this work, we propose a novel inference technique that satisfies all four core requirements. Our key insight is that the problem of learning conditional rewrite rules can be decomposed into three steps: inference of the latent *underlying forms*, learning the *changes* (rewrites), and learning the *conditions*. Moreover, each of these problems can be encoded as a constrained optimization problem that can be solved efficiently by modern *satisfiability modulo theories* (SMT) solvers [40]. Both the decomposition and the encoding into constraints are contributions of this work. We implement this approach in a system called SYPHON and show that it is capable of generating accurate phonological rules in under a minute and from just 5–30 data points.

2.2 Background and Problem Definition

In this section, we illustrate phonological processes and the problem of phonological rule induction using our running example of English verbs.

2.2.1 Rule-Based Phonology

Phonological features. *Phones* (speech sounds) are described using a feature system that groups similar-sounding phones together. For instance, voiced consonants (consonants produced with vibrating vocal cords, like [z], [d], [b]) possess the features +consonant and +voice, while voiceless consonants (like [s], [t], [p]) possess the features +consonant and −voice. Each phone can be uniquely identified by a feature vector: for example [−voice −strident +anterior −distributed] uniquely identifies the sound [t]. However, some phones may be uniquely identified by several feature vectors, and not all feature vectors correspond to phones (the feature system is redundant). For example, the feature vector [+low +high] does not correspond to any phones, as no phone can have both a raised and a lowered tongue body.

Phonological rules. In rule-based phonology, a phonological process is formalized as a conditional rewrite rule that transforms an *underlying form* of a word (roughly, the unique stored form of the word) into its *surface form* (the word as it is intended to be pronounced). In our English past tense example, the underlying form /zɪpd/—formed by concatenating the stem /zɪp/ and past tense suffix /d/—is transformed into the surface form [zɪpt] by a rule that makes an obstruent voiceless when it occurs after a voiceless obstruent:

$$[-\text{sonorant}] \rightarrow [-\text{voice}] \;/\; [-\text{voice}] \_$$

In general, phonological rules have the form $A \rightarrow B \;/\; L \_ R$, where all of A, B, L, and R are feature vectors. The rule means that any phone that matches A and occurs between two phones that match L and R, respectively, will be rewritten to match B (leaving the features not mentioned by B intact). A is called the *target* of the rule, B is called the *structural change*, and L and R are the left and the right *contexts*.[1] In the example above, the right context is omitted, because it is irrelevant to the rule's application; formally, A, L, and R may each be empty feature

[1] In this work we only consider a subset of *strictly local* k=3 rules [30], where either side of the context is restricted to at most a single phone.

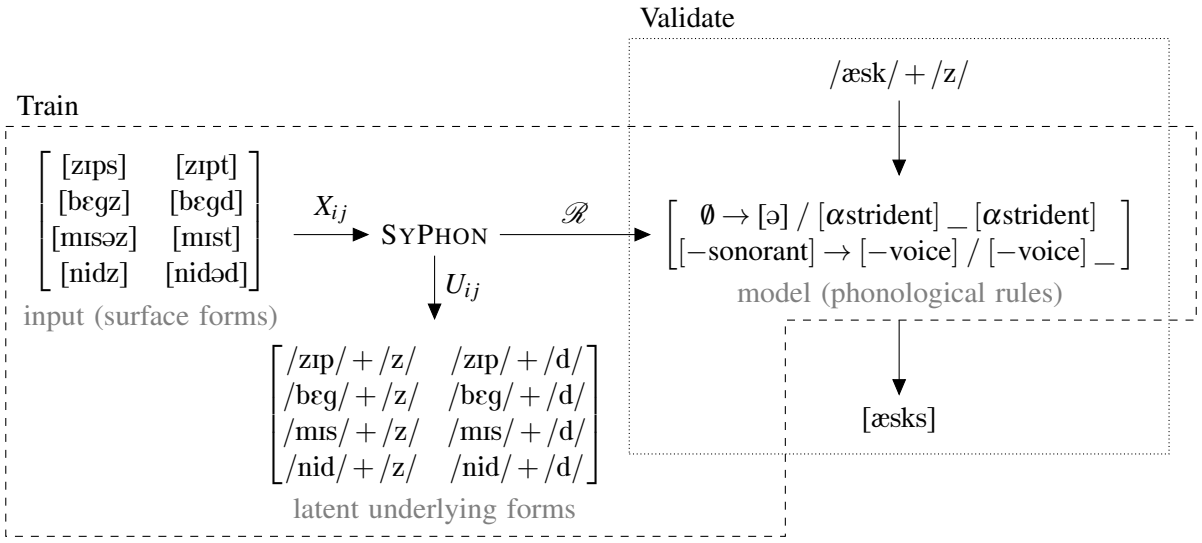


Figure 2.1: The general structure of the problem, shown concretely for English verbs.

vectors, which are defined to match any phone.

Hereafter, we refer to the sequence *LAR* of the target and the context as the *condition* of the rule. If the condition is empty, the rule applies unconditionally. In addition to + and −, the values of features in the condition of the rule may be *variables*, which enforce that features have the same value in different parts of the condition. For example,

$$A \rightarrow B \;/\; [\alpha\text{consonant}] \_ [\alpha\text{consonant}]$$

describes a rule which applies between pairs of consonants and pairs of vowels, but not between a consonant and a vowel.

2.2.2 Problem Definition

The input to our problem is a matrix of surface forms, such as the one shown in Fig. 2.1, left. These forms are arranged into rows, corresponding to different stems, and columns, corresponding to different inflections (in this case, the third-person singular and past tense of English verbs). In the interest of space, we only show four rows from this data set, but a typical input in a phonology textbook problem is only slightly larger and ranges from 5 to 30 rows.

Given these data, our task is to infer the latent underlying forms for each of the words in the input such that the resulting matrix of underlying forms factorizes into stems and suffixes, and to learn a sequence of phonological rules which, when applied elementwise to the matrix of underlying forms, reproduces the matrix of surface forms.

This learned sequence of phonological rules is generative in the following sense: given the underlying form for a new word, such as /æskz/, we can deterministically apply these rules to generate the surface form of that word, [æsks]. We use this property to evaluate the accuracy of the rule set we learned by holding out a portion of the words from the data, and then applying the rule to the underlying forms of those words, which were determined through phonological research.

2.2.3 Phonological Intuition

The design of our system is informed by how linguists solve the problem of phonological rule induction. When a phonologist analyzes these data, they begin by positing underlying forms that are likely to result in the simplest set of rules. For example, they observe that the substring shared in each row is most likely the stem, which surfaces without change; the underlying suffix in the first column in Fig. 2.1 is likely /z/, which sometimes surfaces as [s] and other times as [əz]; and similarly, the underlying suffix in the second column is likely /d/, which can change to [t] or [əd]. The choice of /z/ and /d/ as the underlying suffixes is preferred to, say, /s/ and /t/, because this choice lets us explain all the observed data using only three edits: /z/ $\rightarrow$ [s], /d/ $\rightarrow$ [t], and $\emptyset \rightarrow$ [ə].

The next step is to merge and generalize individual edits: the first two edits are both devoicing an obstruent, so they can be merged into [$-$sonorant] $\rightarrow$ [$-$voice], while the last edit is an insertion and cannot be generalized.

The final step of the analysis is to infer the conditions under which each of the two structural changes occurs. By contrasting examples in the first column, we infer that the insertion happens when the suffix /z/ occurs after a strident (like /s/ in /mɪs/); otherwise, /z/ and /d/

are devoiced whenever they occur after a voiceless obstruent (like /p/ in /zɪp/). The full data set can be explained using the two rules in Fig. 2.1, right. Note that in order to capture the data in both columns, the insertion rule says that [ə] is inserted whenever the stridency of the left and right context matches. Note also that in this case the order of rules matters: for words like /mɪsz/, insertion is applied first, which prevents the devoicing rule from applying.

2.3 Learning Phonological Rules

As illustrated in Fig. 2.1, the input to our learning problem is a matrix of surface forms X_{ij} with I rows and J columns. The goal is to learn a discrete rule set $\mathscr{R}$, while jointly inferring the latent set of I stems S_i and J affixes A_j.

Hypothesis space. The hypothesis space for $\mathscr{R}$ can be formalized as a context-free grammar:

$$\begin{aligned} \mathscr{R} &\Rightarrow R^* & R &\Rightarrow C \rightarrow C \;/\; C \_ C \\ C &\Rightarrow (VF)^* & V &\Rightarrow + \mid - \\ & & F &\Rightarrow \text{consonant} \mid \text{voice} \mid \ldots \end{aligned} \tag{2.1}$$

According to this grammar, $\mathscr{R}$ is a sequence of rules R; each R is defined in terms of four feature vectors C; each feature vector is a sequence of pairs of feature values V and feature names F.

Rewriting. We use $\mathscr{C}_R$ and $\mathscr{B}_R$ to denote the condition and structural change of a rule R, respectively. A feature vector C can be interpreted as a Boolean formula that holds of a phone a if a possesses all features in C; we denote by $|C|$ the number of models of this formula, *i.e.* phones in the inventory Φ that satisfy C. Similarly, $\mathscr{C}_R$ is a Boolean formula over *trigrams* of

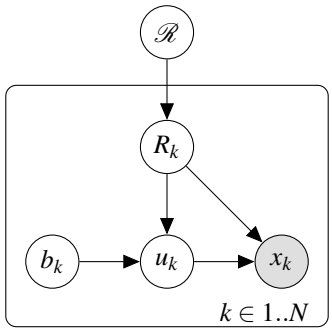


Figure 2.2: Probabilistic model of a phonological process. A rule set $\mathscr{R}$ is sampled from a description length prior. We observe a set of N surface phonemes x_k; each x_k is generated by sampling a rule R_k from $\mathscr{R}$ and an underlying trigram u_k, and deterministically applying R_k to u_k (coin flip b_k decides whether u_k should match R_k's condition).

phones. A *rewrite* of a trigram *abc* by rule R is defined as:

$$R(abc) = \begin{cases} \mathscr{B}_R(b) & \text{if } \mathscr{C}_R(abc) \\ b & \text{otherwise} \end{cases}$$

The notion of rewrites can be extended to words and rule sets.

Learning as constrained optimization. We can now formalize our problem as a hard *correctness constraint* over rules and underlying forms U_{ij}:

$$\mathscr{R}(U_{ij}) = X_{ij} \quad \text{where } U_{ij} \triangleq A_j[S_i] \tag{2.2}$$

Here, $A_j[S_i]$ denotes a concatenation of the prefix/suffix A_j with the stem S_i.

There might be many rule sets $\mathscr{R}$ that are consistent with all the data, and what we would like is to pick one that generalizes to other data that exhibits the same phonological process (for example, the rule inferred in Fig. 2.1 should generalize to other regular English verbs). Hence we frame the learning problem as a constrained optimization problem and derive the objective function using a Bayesian model.

2.3.1 Bayesian Model

Generative process. Intuitively, to generate surface forms X_{ij}, we must sample a single rule set $\mathscr{R}$, I stems S_i, and J affixes A_j, and then deterministically apply $\mathscr{R}$ to each $A_j[S_i]$. Prior work on phonological rule learning [52] assumed that S_i and A_j are sampled uniformly from the language and independently of $\mathscr{R}$. We observe, however, that in most data sets of interest, underlying forms are in fact sampled to contrast the contexts in which $\mathscr{R}$ does and does not apply. We model this intuition as a *strong sampling* process depicted in Fig. 2.2.

For simplicity, in this model each observation corresponds to an individual rule application to an underlying trigram u that produces a surface phoneme x. For example, the rewrite /zɪpz/ → [zɪps] is represented as four observations: /#zi/ → [z], /zɪp/ → [ɪ], /ipz/ → [p], and /pz#/ → [s] (where # encodes word boundary).

Our generative process first samples a ruleset $\mathscr{R}$ from the description length prior over the hypothesis space (2.1):

$$P(\mathscr{R}) \propto 2^{-w_s \cdot \Sigma_{R \in \mathscr{R}} \ell(R)}$$

where $\ell(R)$ is the length of rule R and $w_s > 0$ is a model hyperparameter. For each observation $k \in 1..N$, we pick a rule R_k uniformly from $\mathscr{R}$. Before sampling the underlying trigram u_k, we flip a coin b_k to decide whether we want to sample a *positive* or a *negative* trigram, *i.e.* whether $\mathscr{C}_{R_k}(u_k)$ should hold true; we then sample u_k uniformly from the set of all positive (resp. negative) trigrams (subject to the hard constraint that they form a factorizable matrix U_{ij}). Finally, we deterministically compute $x_k \triangleq R_k(u_k)$. Hence we can define:

$$P(x_k, u_k \mid R_k) = \begin{cases} 0 & \text{if } R_k(u_k) \neq x_k \\ \frac{P(b_k = \top)}{|\mathscr{C}_{R_k}|} & \text{if } \mathscr{C}_{R_k}(u_k) \\ \frac{P(b_k = \bot)}{|\neg\mathscr{C}_{R_k}|} & \text{otherwise} \end{cases}$$

Our goal is to maximize

$$P(\mathscr{R}, R_1, ..., R_N, u_1, ..., u_N \mid x_1, ..., x_N)$$
$$\propto P(\mathscr{R}) \prod_{k=1}^{N} P(x_k, u_k \mid R_k) P(R_k \mid \mathscr{R})$$

Objective function. Taking logs, we can derive the following approximate minimization objective for our constrained optimization problem:

$$w_s \sum_{R \in \mathscr{R}} \ell(R) + N_R^+ \cdot \log(|\mathscr{C}_R|) \tag{2.3}$$

where N_R^+ is the number of positive examples for this rule. (Note that this objective ignores $P(R_k \mid \mathscr{R})$ and b_k, which are assumed to be uniform. It also ignores the negative examples. This provides a reasonable approximation, under the assumption that $|\neg\mathscr{C}_R| \gg |\mathscr{C}_R|$ for each rule R, which holds in the current setting.) This function includes a *simplicity term*, which favors rules with shorter (and hence, more general) conditions, and a *likelihood term*, which favors more specific conditions if there are sufficient positive examples to support them. This likelihood term stems from our strong sampling assumption; we demonstrate its importance for inferring accurate rules in Sec. 2.5.

2.3.2 Inference by Program Synthesis

To solve the constrained optimization problem we build upon a technique from programming languages called *constraint-based program synthesis* [151].

Constraint-based synthesis. The input to (inductive) program synthesis is a DSL that defines the space of possible programs and a set of input-output examples $E = \overrightarrow{\langle i, o \rangle}$; the goal is to find a program whose behavior is consistent with the examples. In constraint-based synthesis, this search problem is reduced to solving a boolean constraint. To this end, we index the DSL by a bitvector c, called a *control vector*. We then define a mapping from control vectors to program

behaviors via an *evaluation relation* $\varphi(c, y, z)$—a boolean formula that holds if and only if a program indexed by c produces output z on input y. Given the evaluation relation, the synthesis problem reduces to solving the following boolean constraint:

$$\exists c. \bigwedge_{\langle i,o\rangle \in E} \varphi(c, i, o)$$

An SMT solver [40] is then used to find a satisfying assignment for c, which allows us to recover the corresponding program. For this approach to succeed, the evaluation relation has to be designed carefully so that it only uses constraints that the solver can efficiently reason about.

Synthesis of phonological rules. In our setting, the DSL is the space of all rule sets $\mathscr{R}$ (up to a certain size), and the evaluation relation $\varphi(c, U, X)$ is the correctness condition (2.2). Importantly, our setting differs from traditional program synthesis in two ways: first, we have to search for both the control vector and the inputs, and second, in addition to satisfying the correctness condition, we also seek to minimize the objective function (2.3). If we encode the objective function as $\psi(c, \overrightarrow{\langle U, X\rangle})$, we can reduce rule learning to the following constrained optimization:

$$\begin{aligned} \textbf{minimize} \quad & \psi(c, \overrightarrow{\langle A_j[S_i], X_{ij}\rangle}) \\ \textbf{subject to} \quad & \bigwedge_{i,j=1,1}^{N,M} \varphi(c, A_j[S_i], X_{ij}) \end{aligned}$$

Given a proper encoding of φ and ψ, this constraint can be solved by an optimizing SMT solver [24]; this is the approach used in prior work [52]. However, this is a very computationally intensive problem. The reason is the astronomical size of the search space: for a problem of factorizing a 10×2 matrix X_{ij} into stems of length $\ell_S = 3$ and affixes of length $\ell_A = 2$, if we limit the maximum number of rules $N_{\mathscr{R}}$ to 2 and consider an inventory Φ with 90 phones and a feature set F with 30 features, we can estimate the size of the search space as $3^{|F|N_{\mathscr{R}}}|\Phi|^{I\ell_S + J\ell_A} \approx 2^{600}$.

Decomposition. To achieve scalable inference, we decompose the global constrained optimization problem into three steps, inspired by phonological intuition we described in Sec. 2.2.3:

1. *Underlying form inference.* In the first step we use an SMT solver to generate likely underlying stems and suffixes. We rank them based on the heuristic that underlying forms U_{ij} that have a smaller edit distance from surface forms X_{ij} are more likely to produce simple rules (Sec. 2.3.3).

2. *Change inference.* Given the set of edits between each U_{ij} and the corresponding X_{ij}, we identify the smallest set of structural changes B that can describe all the edits (Sec. 2.3.4).

3. *Condition inference.* Finally, for each structural change B, we use program synthesis to infer the condition under which this change occurs (Sec. 2.3.5). If this step fails, we go back to step 1 and generate the next candidate matrix U_{ij}.

In the rest of this section we detail these three steps. For illustration purposes, in all examples we will assume that our feature set has just three features: voice v, sonorant s, and continuant c.

2.3.3 Underlying Form Inference

The input to this step is the matrix of surface forms X_{ij} and the output is a set of aligned pairs $\langle U, X \rangle_{ij}$. Tab. 2.1 illustrates this for a 2×2 matrix of English verbs. For example, $\langle U, X \rangle_{11} = \langle [\text{zɪpz}], [\text{zɪps}] \rangle$; we use red to show alignment information (in this case, a single substitution $z \rightarrow s$). Insertions and deletions are represented by alignment with null segments.

Table 2.1: Underlying form inference on English verbs.

Input		Output	
[zɪps]	[zɪpt]	[zɪpz]	[zɪpd]
		[zɪps]	[zɪpt]
[nidz]	[nidəd]	[nidz]	[nid_d]
		[nidz]	[nidəd]

The output matrix $\langle U, X \rangle_{ij}$ has to satisfy two properties: (i) the matrix U_{ij} can be factorized into stems S_i and affixes A_j, and (ii) each pair $\langle U, X \rangle$ has a small edit distance. Our intuition is that underlying forms that have a small edit distance from surface forms are likely to produce simpler rules. Hence we generate candidate matrices $\langle U, X \rangle_{ij}$ in the order of increasing edit distance, until rule inference succeeds for one of them. This strategy will always eventually find a matrix of underlying forms which can be related to the surface forms by a rule set we can infer as long as one exists. This process is not guaranteed to find the global minimum of the objective function (2.3), but we show empirically that it produces good results.

We can encode the properties (i) and (ii) as a boolean constraint over unknown strings with concatenation and length, which can be solved efficiently by the Z3STR2 solver [188]. From the solutions for those unknowns it is straightforward to recover not only the stems and suffixes, but also the required alignment information between the underlying and surface forms.

2.3.4 Change Inference

The input to change inference is the set of all edits in the aligned pairs $\langle U, X \rangle_{ij}$, computed in the previous step, and the output is a set of structural changes that captures all the edits. Tab. 2.2 illustrates this for the edits from Tab. 2.1; columns LHS and RHS show relevant features of the left- and right-hand sides of the edit.

Table 2.2: Change inference on English verbs.

Edit	LHS	RHS	Change
/z/ → [s]	[+v −s +c]	[−v −s +c]	[−v]
/d/ → [t]	[+v −s −c]	[−v −s −c]	
∅ → [ə]	∅	[ə]	[ə]

For each edit, we compute the set of all possible structural changes which are consistent with the edit. For example, the edit /z/ → [s] is consistent with four possible changes: [−v], [−v −s] [−v +c], and [−v −s +c]. Next, we greedily merge change-sets of different edits if their intersection is nonempty. This merging step allows us to identify a small set of distinct

structural changes which together describe all the edits. For example, the change-sets of the first two edits in Tab. 2.2 can be merged to produce the change-set: $\{[-\text{v}], [-\text{v}\ -\text{s}]\}$. The third edit in Tab. 2.2 is an insertion, which changes the values of *all* features present in [ə], and hence cannot be merged. When no more merges are possible, we pick the simplest change from every change set (in this case, we end up with changes $B_1 = [-\text{v}]$ and $B_2 = [\text{ə}]$). This greedy process bounds the maximum number of rules to the number of change sets.

2.3.5 Condition Inference

For each structural change B inferred in the previous step, we now attempt to determine the condition *LAR* under which the change applies. If successful, a rule $A \rightarrow B \ / \ L \_ \ R$ is added to the inferred rule set $\mathscr{R}$; otherwise we go back to underlying form inference and try the next candidate matrix U_{ij}.

For a given change B, the input to condition inference is the set of pairs $\langle u, \ell \rangle_k$, where u_k is a phone trigram in some underlying form and the label ℓ_k can be *positive* ($\top$), *negative* ($\bot$), or *unknown* (?). Tab. 2.3 illustrates this for trigrams from $U = /\text{zɪpz}/$. A trigram is labeled positive

Table 2.3: Input to condition inference for change $[-\text{v}]$ on /zɪpz/ → [zɪps]

u	ℓ	Features
/#zɪ/	$\bot$	[+#] [+v −s][+v +s]
/zɪp/	$\bot$	[+v −s][+v +s][−v −s]
/ɪpz/	?	[+v +s][−v −s][+v −s]
/pz#/	$\top$	[−v −s][+v −s] [+#]

if its middle phone undergoes the change B in the data, negative if it does not undergo B, and unknown if B has no effect on this phone. In our example, neither /ɪ/ nor /p/ in /zɪps/ actually changed, however /zɪp/ is labeled $\bot$ while /ɪpz/ is labeled ?, because /p/ is already $[-\text{v}]$, and hence devoicing has no effect on it. Our goal is to infer a condition consistent with the labels of all the positive and negative trigrams (unknown trigrams are ignored).

Inference by program synthesis. To frame condition inference as a program synthesis

problem we need to define the control vector that indexes the space of all possible conditions, and a corresponding evaluation relation. In our control vector, for each feature f, we use six control variables, which represent the three positions that a feature can appear in a rule (left context, target, and right context) and the two values it can take on ($+$ and $-$). We denote these variables by f_p^v for v in $V = \{+,-\}$ and p in $P = \{l,t,r\}$.

Our evaluation relation takes the form $\varphi(c,u,\ell) \triangleq \text{matches}(c,u) \Leftrightarrow \ell$, where matches is a relation specifying whether the condition indexed by c matches the trigram u. The matches relation is further defined as follows:

$$\text{matches}(c,u) \triangleq \bigwedge_{(f,v,p)\in F\times V\times P} f_p^v \Rightarrow (u_{p,f} = v)$$

where $u_{p,f}$ is the value of feature f at position p in trigram u.

2.3.6 Inductive Bias

In addition to being consistent with the data, we also want the condition to minimize the objective function (2.3). We encode the objective function as

$$w_s s(c) + l(c),$$

where $s(c)$ encodes the simplicity of the condition indexed c (its size), $l(c)$ encodes the likelihood, and w_s is a model hyperparameter which determines the relative importance of simplicity.

The challenge is to encode the likelihood term in a solver-friendly way. To count the number of models of $|\mathscr{C}_R|$, we observe that $|\mathscr{C}_R| = |\mathscr{C}_R^l||\mathscr{C}_R^t||\mathscr{C}_R^r|$, *i.e.* we can independently count the models of the target, and the left and right contexts, so

$$l(c) \triangleq N^+ \sum_{p\in P} \log(|\mathscr{C}_R^p|)$$

We also observe that $|\mathscr{C}_R^p|$ can be encoded efficiently using a constraint whose size is *linear* in

the size of the phone inventory Φ:

$$|\mathscr{C}_R^p| \triangleq \sum_{a \in \Phi} \text{if } \Big(\bigwedge_{\substack{(f,v) \in F \times V \\ a_f \neq v}} \neg f_p^v \Big) \text{ then 1 else 0}$$

Finally, as the solver does not support logarithms, we encode log using a lookup table. This is tractable, since we only need to evaluate the log of each $|\mathscr{C}_R^p|$, which is at most the size of our inventory, roughly 100 phones.

2.3.7 Current limitations

SYPHON currently leverages three simplifying assumptions about rules for domain-specific problem decomposition and SMT encoding, which are crucial to making learning computationally tractable.

Conjunctive conditions. Rule conditions are conjunctions of equalities over feature values, and each rule has a unique change. We can thus decompose the learning process into change inference and condition inference: change inference greedily groups all observed edits into changes, and from then on we assume that each change uniquely corresponds to a rule.

Local context. The condition of each rule is only a function of the target phone and two surrounding phones. This allows us to encode condition inference as learning a formula over *trigrams* of phones, which has a compact encoding as SMT constraints.

Rule interaction. One rule's change does not create conditions for another. This allows us to perform condition inference for each rule independently.

Many attested patterns in real languages go beyond these limitations. We believe that it is possible to lift these restrictions, and still leverage the structure of conditional rewrite rules to retain most of the benefits of our problem decomposition. We leave this extension to future work.

2.4 Data

We evaluate our system on two broad categories of datasets: lexical databases and textbook problems.

2.4.1 Lexical Databases

We use large lexical databases to investigate two (morpho)phonological processes in English: flapping (6457 rows) and regular verb inflections (2756 rows). We process the CMU pronouncing dictionary [172] to create underlying and surface form pairs exemplifying flapping, as in [69]. For verb inflections, we combine morphological information extracted from CELEX-2 [17] with CMU transcriptions to create a database of regular verbs, where each row of the database contains the third-person singular present tense wordform and past tense wordform for a given verb. For both datasets we have gold standard solutions for both rule sets and underlying forms, provided by one of our coauthors, who is a phonologist.

2.4.2 Textbook Problems

For this category, we curated a set of 34 problems from phonology textbooks [80, 121, 138] by selecting problems with local, non-interacting rules. For each problem, the input data set is tailored (by the textbook author) to illustrate a particular phonological process, and contains 20-50 surface forms. For all of these problems we have gold standard solutions, either provided with the textbook or inferred by a phonologist. Gold standard solutions range in complexity from one to two rules. Out of the 34, 21 problems are shared with [47], which we use as the baseline for inference times.

Following the textbooks, these problems are further subdivided into 10 *matrix problems*, 20 *alternation problems*, and 4 *supervised problems*. The matrix problems follow the format presented in Sec. 2.2. The alternation and supervised problems are easier, as we are given more information about the underlying form. For alternation problems, we are essentially given a set of

choices for what the underlying form might be, and for supervised problems the underlying form is given exactly. These problems include examples of phones in complementary distribution. Our problem decomposition allows us to switch out underlying forms inference to handle different kinds of input. According to this classification, the flapping lexical database is an alternation problem and verbs is a matrix problem.

2.5 Experiments

We evaluate our system on the two categories of data sets discussed in Sec. 2.4. We split the 34 textbook problems into 24 development and 10 test problems. Our system has several free parameters (most importantly, the simplicity weight w_s). These were hand-tuned on all of the data except the test problems. For the test problems, we only added missing sounds to the inventory as needed. The 10 test problems are all alternation problems. We leave for future work the investigation of these hyperparameter settings on new matrix problems.

2.5.1 Lexical Database Experiments

We evaluate our system on two large English datasets, one demonstrating flapping, and the other verbs. For each dataset, we learn a rule set from 20, 50 and 100 data points, and evaluate its accuracy on the remaining data. We also perform a syntactic comparison of the rule set against the gold standard rules, which we report as average precision and recall among the sets of features in the two rules. Finally, we compare the latent underlying forms we inferred for each problem to the known correct underlying forms. Tab. 2.4 summarizes the results. Tab. 2.5 (rows 1–3) shows the actual rules inferred on the three flapping training sets.

To examine the importance of likelihood in our system, we repeat this experiment for a variant of our system SYPHON-, which does not consider likelihood and simply optimizes our simplicity prior. As the number of data points increases, the effect of the likelihood grows, and so for SYPHON the recall compared to the gold standard quickly climbs. By contrast, the recall of SYPHON- plateaus, which shows that likelihood is indeed important for finding good rules.

Table 2.4: Accuracy results for the English flapping and verbs corpora data sets on 20, 50 and 100 training examples. SYPHON (SP) and SYPHON- (SP-) are two variants of our model, with and without likelihood, resp. Accuracy reports the generalization accuracy on unseen inputs, rule match and UF indicate how well the inferred rule and underlying form resp. match the gold standard.

	Accuracy		Rule Match				UF
			Precision		Recall		
	SP	SP-	SP	SP-	SP	SP-	
Flap 20	76	52	50	66	31	25	100
Flap 50	93	79	86	71	86	71	100
Flap 100	100	79	100	71	100	71	100
Verb 20	86	73	48	42	83	61	100
Verb 50	88	78	52	50	92	80	100
Verb 100	95	81	62	58	100	82	100

2.5.2 Textbook Problem Experiments

We evaluate the textbook problems under the following three experimental conditions. First, to evaluate the generalization accuracy for unseen inputs, for each of the problems, we hold out a randomly sampled 20% of the data. We then learn a rule set on the remaining data, and validate it against the held out data. We repeat this process three times, and report the average accuracy for each class of problems in Tab. 2.6. We also evaluate syntactic accuracy of the rules and of underlying forms, in the same way as for the lexical databases. Additionally, we evaluate our system on 10 test problems, which were left out entirely when tuning the system hyperparameters. We report the same metrics for these problems. Tab. 2.5 shows inferred rules for selected development problems (rows 4–8) and test problems (rows 9–13).

The accuracy of our inferred rules and underlying forms is 100% for all textbook problems. This is not surprising: the combination of hard constraints and a restrictive DSL makes inferring incorrect rules or underlying forms very difficult. More interesting is the syntactic comparison to the gold standard. This measure is intended to estimate how well the rules SYPHON produces match phonologists' intuition. The results in Tab. 2.6 confirm that without the likelihood term, inference tends to exclude important features from the rule condition: the recall for held out problems goes down by 21%.

Table 2.5: Selected inferred rules: English flapping trained on 20, 50 and 100 examples (1–3); textbook development problems (4–8); textbook test problems (9–13).

Data Set		Inferred Rule
1	Flap 20	$\begin{bmatrix} +\text{cor} \\ -\text{voi} \end{bmatrix} \rightarrow [+\text{approx}] \,/\, [+1\text{stress}]\_$
2	Flap 50	$\begin{bmatrix} +\text{cor} \\ -\text{voi} \\ -\text{del. rel.} \end{bmatrix} \rightarrow \begin{bmatrix} +\text{voi} \\ +\text{approx} \end{bmatrix} \,/\, [+\text{stress}]\_[+\text{syll}]$
3	Flap 100	$\begin{bmatrix} +\text{ant} \\ -\text{voi} \\ -\text{del. rel.} \end{bmatrix} \rightarrow \begin{bmatrix} +\text{voi} \\ +\text{approx} \end{bmatrix} \,/\, [+\text{stress}]\_[+\text{syll}]$
4	Russian	$[-\text{son}] \rightarrow [-\text{voi}] \,/\, \_\,\#$
5	Scottish	$[+\text{syll}] \rightarrow [+\text{long}] \,/\, \_ \begin{bmatrix} +\text{cons} \\ +\text{voi} \\ +\text{cont} \end{bmatrix}$
6	Korean	$\begin{bmatrix} -\text{cont} \\ -\text{voi} \end{bmatrix} \rightarrow \begin{bmatrix} -\text{c.g.} \\ -\text{s.g.} \end{bmatrix} \,/\, \_\,[+\text{c.g.}]$
7	Farsi	$\begin{bmatrix} -\text{cont} \\ +\text{dors} \end{bmatrix} \rightarrow \emptyset \,/\, [+\text{ATR}]\_\,\#$
8	Hungarian	$[-\text{son}] \rightarrow [\alpha\text{voi}] \,/\, \_ \begin{bmatrix} \alpha\text{voi} \\ -\text{del. rel.} \end{bmatrix}$
9	Kishambaa	$[+\text{nas}] \rightarrow [-\text{voi}] \,/\, \_\,[+\text{s.g.}]$
10	Persian	$\begin{bmatrix} +\text{approx} \\ -\text{voi} \end{bmatrix} \rightarrow [+\text{voi}] \,/\, \_\,[-\text{nas}]$
11	Ganda	$[+\text{lat}] \rightarrow [+\text{cont}] \,/\, \begin{bmatrix} -\text{lab} \\ +\text{ATR} \end{bmatrix} \_$
12	Limbu	$\begin{bmatrix} +\text{back} \\ +\text{syll} \end{bmatrix} \rightarrow [+\text{rnd}] \,/\, \begin{bmatrix} +\text{lab} \\ -\text{cont} \end{bmatrix} \_$
13	Kongo	$\begin{bmatrix} -\text{son} \\ +\text{cor} \end{bmatrix} \rightarrow \begin{bmatrix} -\text{ant} \\ +\text{dist} \\ +\text{strid} \end{bmatrix} \,/\, \_ \begin{bmatrix} -\text{rnd} \\ +\text{high} \end{bmatrix}$

Table 2.6: Accuracy of textbook problems. We use (-) for supervised problems without underlying form inference.

	Accuracy		Rule Match				UF
			Precision		Recall		
	SP	SP-	SP	SP-	SP	SP-	SP
MAT	100	100	70	69	77	69	100
ALT	100	100	66	61	71	62	100
SUP	100	100	63	60	71	64	-
TEST	100	100	54	52	61	48	100

Table 2.7: Comparison of the inference times of textbook problems with baseline. We report the median execution times and geometric mean of the speedups. N/A indicates examples where baseline results are unavailable.

	Inference Time (secs)		
	SYPHON	Baseline	Speedup
MAT	30.0	3100	124.6
ALT	10.7	N/A	N/A
SUP	5.3	6333	378.3
TEST	8.3	N/A	N/A

Finally, we compare inference times of SYPHON with the prior work of [47], which is also based on constraint-based program synthesis but does not perform problem decomposition, instead using the global encoding outlined in Sec. 2.3.2. As shown in Tab. 2.7, the decomposition makes SYPHON at least *two orders of magnitude* faster, with an average inference time of just 30 seconds for matrix problems, thus enabling phonologists to use the tool interactively.

2.6 Related Work

Learning (morpho-)phonology is a rich and active area of research; in this overview, we focus on approaches that share our goal of inferring fully interpretable models of phonological processes.

Most closely related to ours is the work of [52] and its follow-up [48] on using program synthesis to infer phonological rules. As mentioned above, the main difference is that SYPHON is two orders of magnitude faster than their system thanks to a novel decomposition and efficient

SMT encoding. On the other hand, we impose extra restrictions on the hypothesis space (*i.e.* we only support local rules), which means that SYPHON is unable to solve some of the harder textbook problems that [48] can solve. In addition, [48] propose a method for inducing phonological representations which are universal across languages.

Beyond program synthesis, [134] use a comparable description length-based approach to unsupervised joint inference of underlying phonological forms and rewrite rule representations of phonological processes, but use a genetic algorithm to find approximate solutions. [69] and [30] discuss supervised learning of restricted classes of finite-state transducer representations of several phonological processes (including English flapping). To date, such work either requires thousands of training observations [69] or has used abstracted and greatly simplified symbol inventories and training data [30].

[82], [72], and [65] propose different methods for learning probabilistic models of phonotactics, which represent gradient co-occurrence restrictions between surface segments within a word. Unlike the current implementation of SYPHON, these models include representational structures that enable them to capture certain non-local phenomena. However, because these models focus on phonotactics, they do not infer underlying forms or rules which relate underlying forms to surface forms.

Finally, much work has focused on learning representations of phonological processes as mappings that minimally violate a set of ranked or weighted constraints [132, 103], but such work has generally taken the constraint definitions as given and focused on learning rankings or weights [73, 157, 25, see *e.g.*], with some exceptions [42, 43].

2.7 Conclusion

We have presented a new approach to learning fully interpretable phonological rules from sets of related surface forms. We have shown that our approach produces rules that largely match linguists' intuition from a handful of examples and within minutes. The contributions of this

paper are a novel decomposition of the global inference problem into three local problems, as well as an encoding of these problems into constraints that can be efficiently solved by an SMT solver.

2.8 Acknowledgements

Chapter 2, in full, is a reprint of the material as it appears in Constraint-based Learning of Phonological Processes. Barke, Shraddha; Kunkel, Rose; Polikarpova, Nadia; Meinhardt, Eric; Baković, Eric; and Bergen, Leon. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. EMNLP-IJCNLP 2019. The dissertation author was a primary investigator and author of this paper.

Chapter 3

Just-in-Time Learning for Bottom-Up Enumerative Synthesis

3.1 Introduction

Consider the task of writing a program that satisfies examples in Fig. 3.1. The desired program must return the substring of the input string `s` on different sides of the dash, depending on the input integer `n`. The goal of *inductive program synthesis* is to perform this task automatically, *i.e.* to generate programs from observations of their behavior.

Input		**Output**
`s`	`n`	
`"1/17/16-1/18/17"`	`1`	`"1/17/16"`
`"1/17/16-1/18/17"`	`2`	`"1/18/17"`
`"01/17/2016-01/18/2017"`	`1`	`"01/17/2016"`
`"01/17/2016-01/18/2017"`	`2`	`"01/18/2017"`

Figure 3.1: Input-output example specification for the `pick-date` benchmark

Inductive synthesis techniques have made great strides in recent years [124, 60, 56, 55, 146, 76, 165], and are powering practical end-user programming tools [75, 99, 86]. These techniques adopt different approaches to perform search over the space of all programs from a *domain-specific language* (DSL). The central challenge of program synthesis is scaling the search to complex programs: as the synthesizer considers longer programs, the search space grows astronomically large, and synthesis quickly becomes intractable, despite clever pruning

strategies employed by state-of-the-art techniques.

For example, consider the following solution to the `pick-date` problem introduced above, using the DSL of a popular synthesis benchmarking platform SyGuS [9]:

```
(substr s (indexof (concat "-" s) "-" (- n 1)) (indexof s "-" n))
```

This solution extracts the correct substring of `s` by computing its starting index `(indexof (concat "-" s) "-" (- n 1))` to be either zero or the position after the dash, depending on the value of `n`. At size 14, this is the shortest SyGuS program that satisfies the examples in Fig. 3.1. Programs of this complexity already pose a challenge to state-of-the art synthesizers: none of the SyGuS synthesizers we tried were able to generate this or comparable solution within ten minutes[1].

Guiding Synthesis with Probabilistic Models. A promising approach to improving the scalability of synthesis is to explore *more likely programs first*. Prior work [102, 19, 116, 49] has proposed guiding the search using different types of learned probabilistic models. For example, if a model can predict, given the input-output pairs in Fig. 3.1, that `indexof` and `substr` are more likely to appear in the solution than other string operations, then the synthesizer can focus its search effort on programs with these operations and find the solution much quicker. Making this approach practical requires solving two major technical challenges: (1) *how to obtain a useful probabilistic model*? and (2) *how to guide the search* given a model?

Learning a Model. Existing approaches [135, 23, 102] are able to learn probabilistic models of code automatically, but require significant amounts of high-quality training data, which must contain hundreds of meaningful programs *per problem domain* targeted by the synthesizer. Such datasets are generally difficult to obtain.

To address this challenge, we propose *just-in-time learning*, a novel technique that learns

[1] CVC4 [136] is able to generate *a* solution within a minute, but its solution overfits to the examples and has size 73, which makes it hard to understand.

a *probabilistic context-free grammar* (PCFG) for a given synthesis problem "just in time", *i.e.* during synthesis, rather than ahead of time. Previous work has observed [146, 127] that partial solutions—*i.e.* programs that satisfy a subset of input-output examples—are often syntactically similar to the final solution. Our technique leverages this observation to collect partial solutions it encounters during search and update the PCFG on the fly, rewarding syntactic elements that occur in these programs. For example, when exploring the search space for the `pick-date` problem, unguided search quickly stumbles upon the short program `(substr s 0 (indexof s "-" n)`, which is a partial solution, since it satisfies two of the four input-output pairs (with $\texttt{n} = 1$). At this point, just-in-time learning picks up on the fact that `indexof` and `substr` seem to be promising operations to solve this problem, boosting their probability in the PCFG. Guided by the updated PCFG, our synthesizer finds the full solution in only 34 seconds.

Guiding the Search. The state of the art in guided synthesis is *weighted enumerative search* using the A^* algorithm, implemented in the EUPHONY synthesizer [102] (see Sec. 3.7 for an overview of other guided search techniques). This algorithm builds upon *top-down enumeration*, which works by gradually filling holes in incomplete programs. Unfortunately, top-down enumeration is not a good fit for just-in-time learning: in order to identify partial solutions, the synthesizer needs to *evaluate* the programs it generates, while with top-down enumeration the majority of synthesizer's time is spent generating incomplete programs that cannot (yet) be evaluated.

To overcome this difficulty, we propose *guided bottom-up search*, a new synthesis algorithm that, unlike prior work, builds upon *bottom-up enumeration*. This style of enumeration works by repeatedly combining small programs into larger programs; every generated program is complete and can be evaluated on the input examples, which enables just-in-time learning to rapidly collect a representative set of partial solutions. In addition, bottom-up enumeration leverages dynamic programming and a powerful pruning technique known as *observational equivalence* [159, 5], which further improves efficiency of synthesis. Our algorithm extends

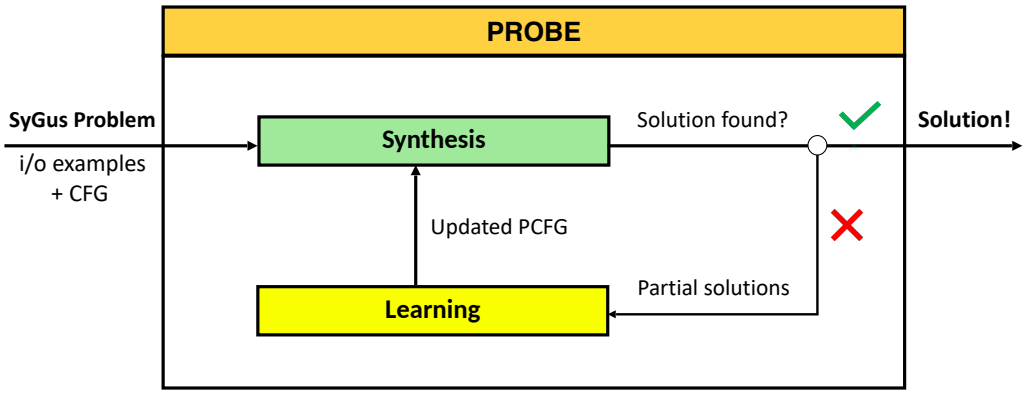


Figure 3.2: Overview of the PROBE system

bottom-up search with the ability to enumerate programs in the order of decreasing likelihood according to a PCFG, and to our knowledge, is the first guided version of bottom-up enumeration. While guided bottom-up search enables just-in-time learning, it can also be used with an independently obtained PCFG.

The* PROBE *Tool. We implemented guided bottom-up search with just-in-time learning in a synthesizer called PROBE. A high-level overview of PROBE is shown in Fig. 3.2. The tool takes as input an inductive synthesis problem in SYGUS format, *i.e.* a context-free grammar of the DSL and a set of input-output examples[2]; it outputs a program from the DSL that satisfies all the examples. Optionally, PROBE can also take as input initial PCFG probabilities suggested by a domain expert or learned ahead of time.

We have evaluated PROBE on 140 SYGUS benchmarks from three different domains: string manipulation, bit-vector manipulation, and circuit transformation. PROBE is able to solve a total of 91 problems within a timeout of ten minutes, compared to only 44 problems for the baseline bottom-up synthesizer and 50 problems for EUPHONY. Note that PROBE outperforms EUPHONY despite requiring no additional training data, which makes it applicable to new

[2]PROBE also supports universally-quantified first-order specifications and reduces them to input-output examples using counter-example guided inductive synthesis (CEGIS) [152]. Since this reduction is entirely standard, in the rest of the paper we focus on inductive synthesis, but we use both kinds of specifications in our evaluation.

ID	Input	Output
e_0	`"a < 4 and a > 0"`	`"a 4 and a 0"`
e_1	`"<open and <close>"`	`"open and close"`
e_2	`"<Change> <string> to <a> number"`	`"Change string to a number"`

Figure 3.3: Input-output example specification for the `remove-angles` benchmark (adapted from [3]).

domains where large sets of existing problems are not available. We also compared PROBE with CVC4 [136], the winner of the 2019 SYGUS competition. Although CVC4 solves more benchmarks than PROBE, its solutions are less interpretable and tend to overfit to the examples: CVC4 solutions are 9 times larger than PROBE solutions on average, and moreover, on the few benchmarks where larger datasets are available, CVC4 achieves only 68% accuracy on unseen data (while PROBE achieves perfect accuracy).

Contributions. To summarize, this paper makes the following contributions:

1. *Guided bottom-up search:* a bottom-up enumerative synthesis algorithm that explores programs in the order of decreasing likelihood defined by a PCFG (Sec. 3.4).

2. *Just-in-time learning:* a new technique for updating a PCFG during synthesis by learning from partial solutions (Sec. 3.5).

3. PROBE*:* a prototype implementation of guided bottom-up search with just-in-time learning and its evaluation on benchmarks from prior work (Sec. 3.6).

3.2 Background

In this section, we introduce the baseline synthesis technique that PROBE builds upon: bottom-up enumeration with observational equivalence reduction [159, 5]. For exposition purposes, hereafter we use a simpler running example than the one in the introduction; the specification for this example, dubbed `remove-angles`, is given in Fig. 3.3. The task is to remove all occurrences of angle brackets from the input string.

S	$\rightarrow$	`arg` | `""` | `"<"` | `">"`	input string and string literals
	|	(`replace` S S S)	`replace s x y` replaces first occurrence of `x` in `s` with `y`
	|	(`concat` S S)	`concat x y` concatenates `x` and `y`

Figure 3.4: A simple CFG for string expressions and the informal semantics of its terminals.

3.2.1 Syntax Guided Synthesis

We formulate our search problem as an instance of *syntax-guided synthesis* (SyGuS) [9]. In this setting, synthesizers are expected to generate programs in a simple language of S-expressions with built-in operations on integers (such as $+$ or $-$) and strings (such as `concat` and `replace`). The input to a SyGuS problem is a syntactic specification, in the form of a context-free grammar (CFG) that defines the space of possible programs and a semantic specification that consists of a set of input-output examples[3]. The goal of the synthesizer is to find a program generated by the grammar, whose behavior is consistent with the semantic specification.

For our running example `remove-angles`, we adopt a very simple grammar of string expressions shown in Fig. 3.4. The semantic specification for this problem is the set of examples $\{e_0, e_1, e_2\}$ from Fig. 3.3. The program to be synthesized takes as input a string `arg` and outputs this string with every occurrence of `"<"` and `">"` removed. Because the grammar in Fig. 3.4 allows no loops or recursion, and the `replace` operation only replaces the first occurrence of a given substring, the solution involves repeatedly replacing the substrings `"<"` and `">"` with an empty string `""`. Fig. 3.5 shows one of the shortest solutions to this problem, which we dub `replace-6`. Note that this benchmark has multiple solutions of the same size that replace `"<"` and `">"` in different order; for our purposes they are equivalent, so hereafter we refer to any one of them as "the shortest solution". The figure also shows two shorter programs, `replace-2` and `replace-3`, which satisfy different subsets of the semantic specification and which we refer to throughout this and next section.

[3] In general, SyGuS supports a richer class of semantic specifications, which can be reduced to example-based specifications using a standard technique, as we explain in Sec. 3.6

ID	Program	Examples Satisfied
replace-2	(replace (replace arg "<" "") ">" "")	$\{e_0\}$
replace-3	(replace (replace (replace arg "<" "") "<" "") ">" "")	$\{e_0, e_1\}$
replace-6	(replace (replace (replace (replace (replace (replace arg "<" "") "<" "") "<" "") ">" "") ">" "") ">" "")	$\{e_0, e_1, e_2\}$

Figure 3.5: Shortest solutions for different example subsets of the `remove-angles` problem.

Height	# Programs	Bank
0	4	`arg, "", "<", ">"`
1	15	`(concat arg arg), (concat arg "<"), (concat arg ">"), (concat "<" "<"), (concat "<" ">"), ...` `(replace arg "<" arg), (replace arg "<" ""), (replace arg "<" ">"), (replace arg ">" "<"), ...`
2	1023	`(concat arg (concat arg arg)), (concat arg (concat ">" ">")), ...(concat "<" (concat arg arg)),` `(concat "<" (replace arg "<" arg)), (concat ">" (concat "<" "<")), (concat ">" (replace arg ">" "<"))`
3	$\sim 30M$	`(concat arg (concat (replace arg "<" arg) arg)), (concat arg (concat (replace arg "<" arg) "<"))` `(concat arg (concat (replace arg "<" arg) ">")), (concat arg (concat (replace arg "<" "") arg)) ...`

Figure 3.6: Programs generated for `remove-angles-short` from the grammar in order of height.

3.2.2 Bottom-up Enumeration

Bottom-up enumeration is a popular search technique in program synthesis, first introduced in the tools TRANSIT [159] and ESCHER [5]. We illustrate this search technique in action using a simplified version of our running example, `remove-angles-short`, where the semantic specification only contains the examples $\{e_0, e_1\}$ (the shortest solution to this problem is the program `replace-3` from Fig. 3.5).

Bottom-up Enumeration. Bottom-up enumeration is a dynamic programming technique that maintains a *bank* of enumerated programs and builds new programs by applying production rules to programs from the bank. Fig. 3.6 illustrates the evolution of the program bank on our running example. Starting with an empty bank, each iteration n builds and adds to the bank all programs of height n. In the initial iteration, we are limited to production rules that require no subexpressions—literals and variables; this yields the programs of height zero: `""`, `"<"`, `">"`, and `arg`. In each following iteration, we build all programs of height $n+1$ using the programs of height up to n as subexpressions. For example at height one, we construct all programs of the form `concat` *x y* and `replace` *s x y*, where $\langle s, x, y \rangle$ are filled with all combinations of height-zero expressions. The efficiency of bottom-up enumeration comes from reusing solutions to overlapping sub-problems, characteristic of dynamic programming: when building a new program, all its sub-expressions are taken directly from the bank and never recomputed.

Observational Equivalence Reduction. Bottom-up synthesizers further optimize the search by discarding programs that are *observationally equivalent* to some program that is already in the bank. Two programs are considered observationally equivalent if they evaluate to the same output for every input in the semantic specification. In our example, the height-one program `(concat arg "")` is not added to the bank because it is equivalent to the height-zero program `arg`. This optimization shrinks the size of the bank at height one from 80 to 15; because each following iteration uses all combinations of programs from the bank as subexpressions, even a small reduction in bank size at lower heights leads to a significant overall speed-up.

Despite this optimization, the size of the bank grows extremely quickly with height, as illustrated in Fig. 3.6. In order to get to the desired program `replace-3`, which has height three, we need to enumerate anywhere between 1024 and $\sim 30M$ programs (depending on the order in which productions and subexpressions are explored within a single iteration). Because of this search space explosion, bottom-up enumerative approach does not find `replace-3` even after 20 minutes.

3.3 Our approach

In this section, we first modify bottom-up search to enumerate programs in the order of *increasing size* rather than *height* (Sec. 3.3.1) and then generalize it to the order of *decreasing likelihood* defined by a probabilistic context-free grammar (Sec. 3.3.2). Finally, we illustrate how the probabilistic grammar can be learned *just in time* by observing partial solutions during search (Sec. 3.3.3).

3.3.1 Size-Based Bottom-up Enumeration

Although exploring *smaller programs first* is common sense in program synthesis, the exact interpretation of "smaller" differs from one approach to another. As we discussed in Sec. 3.2, existing bottom-up synthesizers explore programs in the order of increasing height; at the same time, synthesizers based on other search strategies [9, 97, 13] tend to explore programs

Size	# Programs	Bank
1	4	`arg, "", "<", ">"`
2	0	`None`
3	9	`(concat arg arg), (concat arg "<"), (concat arg ">"), (concat "<" arg),` `(concat "<" "<"), (concat "<" ">"), (concat ">" arg), (concat ">" "<"), (concat ">" ">")`
4	6	`(replace arg "<" arg), (replace arg "<" "), (replace arg "<" ">")` `(replace arg ">" arg), (replace arg ">" ""), (replace arg ">" "<")`
⋮	⋮	⋮
8	349	`(concat (concat (replace arg "<" arg) arg) arg), (concat (replace arg ">" (concat ">" arg)) ">"),` `(replace (concat arg "<") (concat ">" "<") "") ...` `(replace (concat ">" arg) (concat ">" "<") ">")`
9	714	`(concat (concat arg arg) (concat (concat arg arg) arg)), (concat (concat "<" "<") (concat (concat ">" "<") "<")), ...` `(replace (replace arg "<" "") "<" (concat ">" ">")), (replace (replace arg ">" (concat ">" ">")) "<" ">")`
10	2048	`(concat "<" (replace (concat arg arg) (concat ">" arg) "<")),` `...(concat arg (concat (replace arg "<" (concat ">" ">")) ">"))`

Figure 3.7: Programs generated for `remove-angles-short` from the grammar in order of size.

in the order of increasing *size*—*i.e.* total number of AST nodes—rather than height, which has been observed empirically to be more efficient.

To illustrate the difference between the two orders, consider a hypothetical size-based bottom-up synthesizer. Fig. 3.7 shows how the bank would grow with each iteration on our running example. The solution `replace-3` that we are looking for has size ten (and height three). Hence, size-based enumeration only has to explore up to 2048 programs to discover this solution (compared with up to $\sim 30M$ for height-based enumeration). This is not surprising: a simple calculation shows that programs of height three range in size from 8 to 26, and our solution is towards the lower end of this range; in other words, `replace-3` is tall and skinny rather than short and bushy. This is not a mere coincidence: in fact, prior work [143] has observed that *useful programs tend to be skinny rather than bushy*, and therefore exploration in the order of size has a better inductive bias.

Extending Bottom-up Enumeration. Motivated by this observation, we extend the bottom-up enumerative algorithm from Sec. 3.2.2 to explore programs in the order of increasing size. To this end, we modify the way subexpressions are selected from the bank in each search iteration. For example, to construct programs of size four of the form `concat` x y, we only replace $\langle x, y \rangle$ with pairs of programs whose sizes add up to three (the `concat` operation itself takes up one AST node). This modest change to the search algorithm yields surprising efficiency improvements: our size-based bottom-up synthesizer is able to solve the `remove-angles-short` benchmark in only one second! (Recall that the baseline height-based synthesizer times out after

20 minutes).

Unfortunately, the number of programs in the bank still grows exponentially with program size, limiting the range of sizes that can be explored efficiently: for example, the solution to the original `remove-angles` benchmark (`replace-6`) has size 19, and size-based enumeration is unable to find it within the 20 minute timeout. This is where *guided bottom-up search* comes to the rescue.

3.3.2 Guided Bottom-up Search

Previous work has demonstrated significant performance gains in synthesizing programs by exploiting probabilistic models to guide the search [19, 102, 116]. These techniques, however, do not build upon bottom-up enumeration, and hence cannot leverage its two main benefits: reuse of subprograms and observational equivalence reduction (Sec. 3.2.2). Our *first key contribution* is modifying the size-based bottom-up enumeration technique from previous section to guide the search using a *probabilistic context-free grammar* (PCFG). We refer to this modification of the bottom-up algorithm as *guided bottom-up search*.

Probabilistic Context-free Grammars. A PCFG assigns a probability to each production rule in a context-free grammar. For example, Fig. 3.8 depicts a PCFG for our running example that is biased towards the correct solution: it assigns high probabilities to the rules (operations) that appear in `replace-6` and a low probability to the rule `concat` that does not appear in this program. As a result, this PCFG assigns a higher likelihood to the program `replace-6`[4] than it does to other programs of the same size. Hence, an algorithm that explores programs in the order of decreasing likelihood would encounter `replace-6` sooner than size-based enumeration would.

From Probabilities to Discrete Costs. Unfortunately, size-based bottom-up enumeration cannot be easily adapted to work with real-valued probabilities. We observe, however, that the order of program enumeration need not be exact: enumerating *approximately* in the order of decreasing likelihood still benefits the search. Our insight therefore is to convert rule probabilities

[4]The likelihood of a program is the product of the probabilities of all rules involved in its derivation.

			p_R	$-\log(p_R)$	cost_R
S	$\rightarrow$	`arg` | `""` | `"<"` | `">"`	0.188	2.41	2
	|	(`replace` S S S)	0.188	2.41	2
	|	(`concat` S S)	0.059	4.09	4

Figure 3.8: A PCFG for string expressions that is biased towards `replace-6`. For each production rule R, we show its probability p_R and its cost cost_R, which is computed as a rounded negative log of the probability.

into discrete *costs*, which are computed as their rounded negative logs. According to Fig. 3.8, the high-probability rules have a low cost of two, and the low-probability rule `concat` has a higher cost of four. The cost of a program is computed by summing up the costs of its productions, for example:

$$\begin{aligned}\text{cost}(\texttt{concat arg "<"}) &= \text{cost}(\texttt{concat}) + \text{cost}(\texttt{arg}) + \text{cost}(\texttt{"<"}) \\ &= 4 + 2 + 2 = 8\end{aligned}$$

Hence, the order of increasing cost approximately matches the order of decreasing likelihood.

Extending Size-based Enumeration. With the discrete costs at hand, guided bottom-up search is essentially the same as the size-based search detailed in Sec. 3.3.1, except that it takes the cost of the top-level production into account when constructing a new program. Fig. 3.9 illustrates the working of this algorithm. For example, at cost level 8, we build all programs of the form `concat` x y, where the costs of x and y sum up to $8-4=4$. The cost of our solution `replace-6` is 38, which places it within the first 130K programs the search encounters; on the other hand, its size is 19, placing it within the first $\sim 4M$ programs in the order of size. As a consequence, size-based enumeration cannot find this program within 20 minutes, but guided enumeration, given the PCFG from Fig. 3.8, is able to discover `replace-6` within 5 seconds.

3.3.3 Just-in-Time Learning

In the previous section we have seen that guided bottom-up search can find solutions efficiently, given an appropriately biased PCFG. But how can we obtain such a PCFG for

Cost	# Programs	Bank
2	4	`arg, "", "<", ">"`
8	15	`(replace arg "<" arg), (replace arg "<" ""), (replace arg "<" ">")` `(replace arg ">" "<"), (concat "<" arg), (concat "<" "<") ...`
20	1272	`(replace "<" (replace arg (replace arg "<" "") "") ""), (replace "<" (replace arg (replace arg "<" "") "") ">") ...` `(replace (replace arg ">" "<") (replace arg ">" "") arg), (replace (replace arg ">" "<") (replace arg ">" "") ">")`
⋮	⋮	⋮
38	130K	`(str.replace (replace arg "<" (replace (replace arg ">" "<") ">" arg)) (replace (replace arg "<" "") ">" arg) "<")` `(replace (replace arg "<" (replace (replace arg ">" "<") ">" arg)) (replace (replace arg "<" "") ">" arg) ">") ...`

Figure 3.9: Programs generated for `remove-angles` using guided bottom-up search with the PCFG in Fig. 3.8

each synthesis problem? Prior approaches have proposed learning probabilistic models from a corpus of existing solutions [116, 102] (see Sec. 3.7 for a detailed discussion). While achieving impressive results, these approaches are computationally expensive and, more importantly, require high-quality training data, which is generally hard to obtain. Can we benefit from guided search when training data is not available?

Our *second key contribution* is a new approach to learning probabilistic models of programs, which we dub *just-in-time learning*. This approach is inspired by an observation made in prior work [146, 127] that *partial solutions*—programs that satisfy a subset of the semantic specification—often share syntactic similarity with the full solution. We can leverage this insight to iteratively bias the PCFG during synthesis, rewarding productions that occur in partial solutions we encounter.

Enumeration with Just-in-time Learning. We illustrate just-in-time learning on our running example `remove-angles`. We begin enumeration with a uniform PCFG, which assigns the same probability to each production[5]. In this initial PCFG every production has cost 3 (see Fig. 3.10).

With a uniform PCFG, our search starts off exactly the same as size-based search of Sec. 3.3.1. At size 7 (cost level 21), the search encounters the program `replace-2`, which satisfies the example e_0. Since this program contains productions `replace`, `arg`, `""`, `">"`, and `"<"`, we *reward* these productions by decreasing their cost, as indicated in Fig. 3.10; after this update, the cost of the production `concat` does not change, so our solution is now cheaper relative to

[5]The algorithm can also be initialized with a pre-learned PCFG if one is available.

Partial Solution	Examples Satisfied	PCFG costs
	$\emptyset$	`arg`, `""`, `"<"`, `">"`, `replace`, `concat` $\mapsto 3$
`replace-2`	$\{e_0\}$	`arg`, `""`, `"<"`, `">"`, `replace` $\mapsto 2$; `concat` $\mapsto 3$
`replace-3`	$\{e_0, e_1\}$	`arg`, `""`, `"<"`, `">"`, `replace` $\mapsto 2$; `concat` $\mapsto 4$

Figure 3.10: Just-in-time learning: as the search encounters partial solutions that satisfy new subsets of examples, PCFG costs are adjusted and the relative cost of `concat`, which is not present in the solution, increases.

other programs of the same size. With the new PCFG at hand, the enumeration soon encounters another partial solution, `replace-3`, which covers the examples e_0 and e_1. Since this program uses the same productions as `replace-2` and satisfies even more examples, the difference in cost between the irrelevant production `concat` and the relevant ones increases even more: in fact, we have arrived at the same biased PCFG we used in Sec. 3.3.2 to illustrate the guided search algorithm.

Challenge: Selecting Promising Partial Solutions. As this example illustrates, the more partial solutions we encounter that are similar to the final solution, the more biased the PCFG becomes, gradually steering the search in the right direction. The key challenge with this approach is that the search might encounter hundreds or thousands of partial solutions, and many of them have irrelevant syntactic features. In our running example, there are in fact more than 3100 programs that satisfy at least one of the examples e_0 or e_1. For instance, the program

```
replace (replace (replace (concat arg "<") "<" "") "<" "") ">" ""
```

satisfies e_0, but contains the `concat` production, so if we use this program to update the PCFG, we would steer the search away from the final solution. Hence, the core challenge is to identify *promising* partial solutions, and only use those to update the PCFG.

A closer look at this program reveals that it has the same behavior as the shorter program `replace-2`, but it contains an irrelevant subexpression that appends `"<"` to `arg` only to immediately replace it with an empty string! In our experience, this is a common pattern: whenever a

partial solution p' is *larger* than another partial solution p but solves the same subset of examples, then p' often syntactically differs from p by an irrelevant subexpression, which happens to have no effect on the inputs solved by the two programs. Following this observation, we only consider a partial solution p promising—and use it to update the PCFG—when it is one of the *shortest* solutions that covers a given subset of examples.

Powered by just-in-time learning, PROBE is able to find the solution `replace-6` within 23 seconds, starting from a uniform PCFG: only a slight slowdown compared with having a biased PCFG from the start. Note that EUPHONY, which uses a probabilistic model learned from a corpus of existing solutions, is unable to solve this benchmark even after 10 minutes.

3.4 Guided Bottom-up Search

In this section, we describe our guided bottom-up search algorithm. We first formulate our problem of guided search as an instance of an inductive SYGUS problem. We then present our algorithm that enumerates programs in the order of decreasing likelihood.

3.4.1 Preliminaries

Context-free Grammar. A *context-free grammar* (CFG) is quadruple $\mathscr{G} = (\mathscr{N}, \Sigma, \mathscr{S}, \mathscr{R})$, where $\mathscr{N}$ denotes a finite, non-empty set of non-terminal symbols, Σ denotes a finite set of terminals, $\mathscr{S}$ denotes the starting non-terminal, and $\mathscr{R}$ is the set of production rules. In our setting, each terminal $t \in \Sigma$ is associated with an *arity* $\text{arity}(t) \geq 0$, and each production rule $\text{R} \in \mathscr{R}$ is of the form $\text{N} \to (t\ \text{N}_1\ \ldots\ \text{N}_k)$, where $\text{N}, \text{N}_1, \ldots, \text{N}_k \in \mathscr{N}$, $t \in \Sigma$, and $\text{arity}(t) = k$[6]. We denote with $\mathscr{R}(\text{N})$ the set of all rules $\text{R} \in \mathscr{R}$ whose left-hand side is N. A sequence $\alpha \in (\mathscr{N} \cup \Sigma)^*$ is called a *sentential form* and a sequence $s \in \Sigma^*$ is a called a *sentence*. A grammar $\mathscr{G}$ defines a (leftmost) *single-step derivation* relation on sentential forms: $s\text{N}\alpha \Rightarrow s\beta\alpha$ if $\text{N} \to \beta \in \mathscr{R}$. The reflexive transitive closure of this relation is called (leftmost) *derivation* and written $\Rightarrow^*$. All

[6] An astute reader might have noticed that we can formalize this grammar as a *regular tree grammar* instead; we decided to stick with the more familiar context-free grammar for simplicity.

grammars we consider are unambiguous, *i.e.* every sentential form has at most one derivation.

Programs. A *program* P is a sentence derivable from some $\mathrm{N} \in \mathscr{N}$; we call a program *whole* if it is derivable from $\mathscr{S}$. The set of all programs is called the *language* of the grammar $\mathscr{G}$: $\mathscr{L}(\mathscr{G}) = \{s \in \Sigma^* \mid \mathrm{N} \Rightarrow^* s\}$. The *trace* of a program $\mathrm{tr}(P)$ is the sequence of production rules $\mathrm{R}_1, \ldots, \mathrm{R}_n$ used in its derivation ($\mathrm{N} \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow P$). The *size* of a program $|P|$ is the length of its trace. We assign semantics $[\![P]\!]\colon \mathrm{Val}^* \to \mathrm{Val}$ to each program P, where Val is the set of run-time values.

Inductive Syntax-Guided Synthesis. An *inductive* syntax-guided synthesis (SyGuS) problem is defined by a grammar $\mathscr{G}$ and a set of input-output examples $\mathscr{E} = \overrightarrow{\langle i, o \rangle}$, where $i \in \mathrm{Val}^*$, $o \in \mathrm{Val}$[7]. A *solution* to the problem is a program $P \in \mathscr{L}(\mathscr{G})$ such that $\forall \langle i, o \rangle \in \mathscr{E}$, $[\![P]\!](i) = o$. Without loss of generality, we can assume that only whole programs can evaluate to the desired outputs o, hence our formulation need not explicitly require that the solution be whole.

Probabilistic Context-free Grammar. A *probabilistic context-free grammar* (PCFG) $\mathscr{G}_p$ is a pair of a CFG $\mathscr{G}$ and a function $p : \mathscr{R} \to [0, 1]$ that maps each production rule $\mathrm{R} \in \mathscr{R}$ to its probability. Probabilities of all the rules for given non-terminal $\mathrm{N} \in \mathscr{N}$ sum up to one: $\forall \mathrm{N}. \sum_{\mathrm{R} \in \mathscr{R}(\mathrm{N})} p(\mathrm{R}) = 1$. A PCFG defines a probability distribution on programs: a probability of a program is the product of probabilities of all the productions in its trace $p(P) = \prod_{\mathrm{R}_i \in \mathrm{tr}(P)} p(\mathrm{R}_i)$.

Costs. We can define the *real cost* of a production as $\mathrm{rcost}(\mathrm{R}) = -\log(p(\mathrm{R}))$; then the real costs of a program can be computed as $\mathrm{rcost}(P) = -\log(p(P)) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} \mathrm{rcost}(\mathrm{R}_i)$. For the purpose of our algorithm, we define *discrete costs*, which are real costs rounded to the nearest integer: $\mathrm{cost}(\mathrm{R}) = \lfloor \mathrm{rcost}(\mathrm{R}) \rceil$. The cost of a program P is defined as the sum of costs of all the productions in its trace: $\mathrm{cost}(P) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} \mathrm{cost}(\mathrm{R}_i)$.

[7] In general, the SyGuS problem allows first-order formulae as a specification, and prior work has shown how to reduce this general formulation to inductive formulation using CEGIS [13, 102].

3.4.2 Guided Bottom-up Search Algorithm

Algorithm 5 presents our guided bottom-up search algorithm. The algorithm takes as input a PCFG $\mathscr{G}_p$ and a set of input-output examples $\mathscr{E}$, and enumerates programs in the order of increasing discrete costs according to $\mathscr{G}_p$, until it finds a program P that satisfies the entire specification $\mathscr{E}$ or reaches a certain cost limit LIM. The algorithm maintains a search state that consists of (1) the current cost level LVL; (2) program bank B, which stores all enumerated programs indexed by their cost; (3) evaluation cache E, which stores evaluation results of all programs in B (for the purpose of checking observational equivalence); and (4) the set PSol, which stores all enumerated partial solutions. Note that the algorithm returns the current search state and optionally takes a search state as input; we make use of this in Sec. 3.5 to resume search from a previously saved state.

Every iteration of the loop in lines 3–14 enumerates all programs whose costs are equal to LVL. New programs with a given cost are constructed by the auxiliary procedure NEW-PROGRAMS, which we describe below. In line 5, every new program P is evaluated on the inputs from the semantic specification $\mathscr{E}$; if the program matches the specification exactly, it is returned as the solution. Otherwise, if the evaluation result is already present in E, then P is deemed observationally equivalent to another program in B and discarded. A program with new behavior is added to the bank at cost LVL and its evaluation result is cached in E; moreover, if the program satisfies some of the examples in $\mathscr{E}$, it is considered a partial solution and added to PSol.

The auxiliary procedure NEW-PROGRAMS takes in the PCFG $\mathscr{G}_p$, the current cost LVL, and a bank B where all levels below the current one are fully filled. It computes the set of all programs of cost LVL in $\mathscr{G}_p$. For the sake of efficiency, instead of returning the whole set at once, NEW-PROGRAMS is implemented as an *iterator*: it yields each newly constructed program lazily, and will not construct the whole set if a solution is found at cost LVL. To construct a program of cost LVL, the procedure iterates over all production rules $\mathrm{R} \in \mathscr{R}$. Once R is chosen as the top-level production in the derivation of the new program, we have a budget of $\mathrm{LVL} - \mathrm{cost}(\mathrm{R})$ to

Algorithm 1. Guided Bottom-up search algorithm

Input: PCFG $\mathscr{G}_p$, input-output examples $\mathscr{E}$, and optionally, the initial state of the search
Output: A solution P or $\bot$, and the current state of the search

1: **procedure** GUIDED-SEARCH($\mathscr{G}_p, \mathscr{E}, \langle \text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0 \rangle = \langle 0, \emptyset, \emptyset, \emptyset \rangle$)
2: $\quad$ LVL, B, E, PSol $\leftarrow$ $\text{LVL}_0, \text{B}_0, \text{E}_0, \text{PSol}_0$ ▷ Initialize state of the search
3: $\quad$ **while** LVL $\leq \text{LVL}_0 + \text{LIM}$ **do**
4: $\quad\quad$ **for** $P \in$ NEW-PROGRAMS($\mathscr{G}_p$, LVL, B) **do** ▷ For all programs of cost LVL
5: $\quad\quad\quad$ EVAL $\leftarrow [\langle i, [\![P]\!](i) \rangle \mid \langle i, o \rangle \in \mathscr{E}]$ ▷ Evaluate on inputs from $\mathscr{E}$
6: $\quad\quad\quad$ **if** (EVAL $= \mathscr{E}$) **then**
7: $\quad\quad\quad\quad$ **return** $(P, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol} \rangle)$ ▷ P fully satisfies $\mathscr{E}$, solution found!
8: $\quad\quad\quad$ **else if** (EVAL $\in$ E) **then**
9: $\quad\quad\quad\quad$ **continue** ▷ P is observationally equivalent to another program in B
10: $\quad\quad\quad$ **else if** (EVAL $\cap \mathscr{E} \neq \emptyset$) **then** ▷ P partially satisfies $\mathscr{E}$
11: $\quad\quad\quad\quad$ PSol $\leftarrow$ PSol $\cup P$
12: $\quad\quad\quad$ B[LVL] $\leftarrow$ B[LVL] $\cup \{P\}$ ▷ Add to the bank, indexed by cost
13: $\quad\quad\quad$ E $\leftarrow$ E $\cup$ EVAL ▷ Cache evaluation result
14: $\quad\quad$ LVL $\leftarrow$ LVL $+ 1$
15: $\quad$ **return** $(\bot, \langle \text{LVL}, \text{B}, \text{E}, \text{PSol} \rangle)$ ▷ Cost limit reached

Input: PCFG $\mathscr{G}_p$, cost level LVL, program bank B filled up to LVL $- 1$
Output: Iterator over all programs of cost LVL ▷ For all production rules
16: **procedure** NEW-PROGRAMS($\mathscr{G}_p$, LVL, B)
17: $\quad$ **for** $(\text{R} = \text{N} \to (t\ \text{N}_1\ \text{N}_2\ \ldots\ \text{N}_k) \in \mathscr{R})$ **do**
18: $\quad\quad$ **if** cost(R) $=$ LVL $\wedge\, k = 0$ **then** ▷ t has arity zero
19: $\quad\quad\quad$ **yield** t
20: $\quad\quad$ **else if** cost(R) $<$ LVL $\wedge\, k > 0$ **then** ▷ t has non-zero arity
21: $\quad\quad\quad$ **for** $(c_1, \ldots, c_k) \in \left\{ [1, \text{LVL}]^k \,\middle|\, \sum c_i = \text{LVL} - \text{cost(R)} \right\}$ **do** ▷ For all subexpression costs
22: $\quad\quad\quad\quad$ **for** $(P_1, \ldots, P_k) \in \{ \text{B}[c_1] \times \ldots \times \text{B}[c_k] \mid \bigwedge_i \text{N}_i \Rightarrow^* P_i \}$ **do** ▷ For all subexpressions
23: $\quad\quad\quad\quad\quad$ **yield** $(t\ P_1\ \ldots\ P_k)$

allocate between the subexpressions; line 21 iterates over all possible subexpression costs that add up to this budget. Once the subexpression costs $c_1, \ldots, c_k$ have been fixed, line 22 iterates over all k-tuples of programs from the bank that have the right costs and the right *types* to serve as subexpressions: $\text{N}_i \Rightarrow^* P_i$ means that P_i can replace the nonterminal N_i in the production rule R. Finally, line 23 builds a program from the production rule R and the subexpressions P_i.

3.4.3 Guarantees

Soundness. The procedure GUIDED-SEARCH is *sound*: given $\mathscr{G}_p = \langle \mathscr{G}, p \rangle$ and $\mathscr{E}$, if the procedure returns $(P, \_)$, then P is a solution to the inductive SYGUS problem $(\mathscr{G}, \mathscr{E})$. It is

straightforward to show that P satisfies the semantic specification $\mathscr{E}$, since we check this property directly in line 6. Furthermore, $P \in \mathscr{L}(\mathscr{G})$, since P is constructed by applying a production rule R to programs derived from appropriate non-terminals (see check in line 22).

Completeness. The procedure GUIDED-SEARCH is *complete*: if P^* is a solution to the inductive SYGUS problem $(\mathscr{G}, \mathscr{E})$, such that $\text{cost}(P^*) = C$, and $C \leq \text{LVL}_0 + \text{LIM}$, then the algorithm will return $(P, \_)$, where $\text{cost}(P) \leq C$. Completeness follows by observing that each level of the bank is complete up to observational equivalence: if $P \in \mathscr{L}(\mathscr{G})$ and $\text{cost}(P) \leq C$, then at the end of the iteration with $\text{LVL} = C$, either $P \in \text{B}$ or $\exists P' \in \text{B}$ s.t. $\text{cost}(P') \leq \text{cost}(P)$ and $\forall \langle i, o \rangle \in \mathscr{E}$ s.t. $[\![P]\!](i) = [\![P']\!](i)$. This in turn follows from the completeness of NEW-PROGRAMS (it considers all combinations of costs of R and the subexpressions that add up to LVL), monotonicity of costs (replacing a subexpression with a more expensive one yields a more expensive program) and compositionality of program semantics (replacing a subexpression with an observationally equivalent one yields an observationally equivalent program).

Prioritization. We would also like to claim that GUIDED-SEARCH enumerates programs in the order of decreasing likelihood. This property would hold precisely if we were to enumerate programs in order of increasing real cost rcost: since the log function is monotonic, $p(P_1) < p(P_2)$ iff $\text{rcost}(P_1) < \text{rcost}(P_2)$. Instead GUIDED-SEARCH enumerates programs in the order of increasing discrete cost cost, so this property only holds approximately due to the rounding error. Empirical evaluation shows, however, that this approximate prioritization is effective in practice (Sec. 3.6).

3.5 Just in time learning

In this section, we introduce a new technique we call just-in-time learning that updates the probabilistic model used to guide synthesis by learning from partial solutions. We first present the overall PROBE algorithm in Sec. 3.5.1 and then discuss the three steps involved in updating the PCFG in the remainder of the section.

Algorithm 2. The PROBE algorithm

Input: CFG $\mathscr{G}$, set of input-output examples $\mathscr{E}$
Output: A solution P or $\perp$

1: **procedure** PROBE($\mathscr{G}, \mathscr{E}$)
2: $\quad \mathscr{G}_p \leftarrow \langle \mathscr{G}, p_u \rangle$ ▷ Initialize PCFG to uniform
3: $\quad$ LVL, B, E $\leftarrow 0, \emptyset, \emptyset$ ▷ Initialize search state
4: $\quad$ **while** not timeout **do**
5: $\quad\quad P, \langle$LVL, B, E, PSol$\rangle \leftarrow$ GUIDED-SEARCH $(\mathscr{G}_p, \mathscr{E}, \langle$LVL, B, E, $\emptyset\rangle)$ ▷ Search with current PCFG $\mathscr{G}_p$
6: $\quad\quad$ **if** $P \neq \perp$ **then**
7: $\quad\quad\quad$ **return** P ▷ Solution found
8: $\quad\quad$ PSol $\leftarrow$ SELECT(PSol, E) ▷ Select promising partial solutions
9: $\quad\quad$ **if** PSol $\neq \emptyset$ **then**
10: $\quad\quad\quad \mathscr{G}_p \leftarrow$ UPDATE($\mathscr{G}_p$, PSol, E) ▷ Update the PCFG $\mathscr{G}_p$
11: $\quad\quad\quad$ LVL, B, E $\leftarrow 0, \emptyset, \emptyset$ ▷ Restart the search
12: $\quad$ **return** $\perp$

3.5.1 Algorithm Summary

The overall structure of the PROBE algorithm is presented in Algorithm 2. The algorithm iterates between the following two phases until timeout is reached:

1. *Synthesis phase* searches over the space of programs in order of increasing discrete costs using the procedure GUIDED-SEARCH from Sec. 3.4.

2. *Learning phase* updates the PCFG using the partial solutions found in the synthesis phase.

PROBE takes as input an inductive SYGUS problem $\mathscr{G}, \mathscr{E}$. It starts by initializing the PCFG with CFG $\mathscr{G}$ and a uniform distribution p_u, which assigns every production rule $\mathrm{R} = \mathrm{N} \to \beta$ the probability $p(\mathrm{R}) = 1/|\mathscr{R}(\mathrm{N})|$. Each iteration of the **while**-loop corresponds to one synthesis-learning cycle. In each cycle, PROBE first invokes GUIDED-SEARCH with the current search state. If the search finds a solution, PROBE terminates successfully (line 7); otherwise it enters the learning phase, which consists of three steps. First, procedure SELECT selects *promising* partial solutions (line 8); if no such solutions have been found, the search simply resumes from the current state. Otherwise, the second step is to use the promising partial solutions to UPDATE the PCFG (line 10), and the third step is to restart the search (line 11).

These three steps are detailed in the rest of this section.

3.5.2 Selecting Promising Partial Solutions

The procedure SELECT takes as input the set of partial solutions PSol returned by GUIDED-SEARCH, and selects the ones that are *promising* and should be used to update the PCFG. We illustrate this process using the synthesis problem in Fig. 3.11; some partial solutions generated for this problem are listed in Fig. 3.12. The shortest full solution for this problem is:

```
(substr arg (- (indexof arg "-" 3) 3) 3)
```

Objectives. An effective selection procedure must balance the following two objectives.

(a) *Avoid rewarding irrelevant productions:* The reason we cannot simply use *all* generated partial solutions to update the PCFG is that partial solutions often contain irrelevant subprograms, which do not in fact contribute to solving the synthesis problem; rewarding productions from these irrelevant subprograms derails the search. For example, consider P_0 and P_1 in Fig. 3.12: intuitively, these two programs solve the examples $\{e_0, e_1\}$ *in the same way*, but P_1 also performs an extraneous character replacement, which happens to not affect its behavior on these examples. Hence, we would like to discard P_1 from consideration to avoid rewarding the irrelevant production `replace`. Observe that P_0 and P_1 satisfy the same subset of examples but P_1 has a higher cost; this suggests discarding partial solutions that are subsumed by a cheaper program.

(b) *Reward different approaches:* On the other hand, different partial solutions might represent inherently different approaches to solving the task at hand. For example, consider partial solutions P_0 and P_2 in Fig. 3.12; intuitively, they represent different strategies for computing the starting position of the substring: fixed index *vs.* search (`indexof`). We would like to consider P_2 promising: indeed, `indexof` turns out to be useful in the final solution. We observe that although P_2 solves the same number of examples and has a higher cost than P_0, it solves a different *subset*

ID	Input	Output
e_0	`"+95 310-537-401"`	`"310"`
e_1	`"+72 001-050-856"`	`"001"`
e_2	`"+106 769-858-438"`	`"769"`

Figure 3.11: A set of input-output examples for a string transformation (adapted from [3]).

Cycle	ID	Examples Satisfied	Partial Solutions	Cost
1	P_0	$\{e_0, e_1\}$	`(substr arg 4 3)`	20
2	P_1	$\{e_0, e_1\}$	`(replace (substr arg 4 3) " " arg)`	21
3	P_2	$\{e_1, e_2\}$	`(substr arg (indexof arg (at arg 5) 3) 3)`	37
3	P_3	$\{e_1, e_2\}$	`(substr arg (- 4 (to.int (at arg 4))) 3)`	37

Figure 3.12: Partial solutions and the corresponding subset of examples satisfied for the problem in Fig. 3.11

of examples, and hence should be considered promising.

Our goal is to find the right trade-off between the two objectives. Selecting too many partial solutions might lead to rewarding irrelevant productions and more frequent restarts (recall that search is restarted only if new promising partial solutions were found in the current cycle). On the other hand, selecting too few partial solutions might lead the synthesizer down the wrong path or simply not provide enough guidance, especially when the grammar is large.

Selection Schemes. Based on these objectives, we designed three *selection schemes*, which make different trade-offs and are described below from most to least selective. Note that all selection schemes need to preserve information about promising partial solutions between different synthesis-learning cycles, to avoid rewarding the same solution again after synthesis restarts. We evaluate the effectiveness of these schemes in comparison to the baseline (using all partial solutions) in Sec. 3.6.

1. LARGEST SUBSET: This scheme selects *a single cheapest* program (first enumerated) that satisfies *the largest subset* of examples encountered so far across all synthesis cycles. Consequently, the number of promising partial solutions it selects is always smaller than the size of $\mathscr{E}$. Among partial solutions in Fig. 3.12, this scheme picks a single program P_0.

2. FIRST CHEAPEST: This scheme selects *a single cheapest* program (first enumerated)

that satisfies *a unique subset* of examples. The partial solutions $\{P_0, P_2\}$ from Fig. 3.12 are selected by this scheme. This scheme still rewards a small number of partial solutions, but allows different approaches to be considered.

3. ALL CHEAPEST: This scheme selects *all cheapest* programs (enumerated during a single cycle) that satisfy *a unique subset* of examples. The partial solutions $\{P_0, P_2, P_3\}$ are selected by this scheme. Specifically, P_2 and P_3 satisfy the same subset of examples; both are considered since they have the same cost. This scheme considers more partial solutions than FIRST CHEAPEST, which refines the ability to reward different approaches.

3.5.3 Updating the PCFG

Procedure UPDATE uses the set of promising partial solution PSol to compute the new probability for each production rule $\mathrm{R} \in \mathscr{R}$ using the formula:

$$p(\mathrm{R}) = \frac{p_u(\mathrm{R})^{(1-\mathrm{FIT})}}{Z} \qquad \text{where} \qquad \mathrm{FIT} = \max_{\{P \in \mathrm{PSol} \mid \mathrm{R} \in \mathrm{tr}(P)\}} \frac{|\mathscr{E} \cap \mathrm{E}[P]|}{|\mathscr{E}|}$$

where Z denotes the normalization factor, and FIT is the highest proportion of input-output examples that any partial solution derived using this rule satisfies. Recall that p_u is the uniform distribution for $\mathscr{G}$. This rule assigns higher probabilities to rules that occur in partial solutions that satisfy many input-output examples.

3.5.4 Restarting the Search

Every time the PCFG is updated during a learning phase, PROBE restarts the bottom-up enumeration from scratch, *i.e.* empties the bank B (and the evaluation cache E) and resets the current cost LVL to zero. At a first glance this seems like a waste of computation: why not just resume the enumeration from the current state? The challenge is that any update to the PCFG renders the program bank outdated, and updating the bank to match the new PCFG requires the amount of computation and/or memory that does not pay off in relation to the simpler approach

of restarting the search. Let us illustrate these design trade-offs with an example.

Consider again the synthesis problem in Fig. 3.11, and two programs encountered during the first synthesis cycle: the program `0` with cost 5 and the program `(indexof arg "+")` with cost 15. Note that both programs evaluate to 0 on all three example inputs, *i.e.* they belong to the same observational *equivalence class* $[0,0,0]$; hence the latter program is *discarded* by observational equivalence reduction, while the former, discovered first, is chosen as the *representative* of its equivalence class and appears in the current bank B.

Now assume that during the subsequent learning phase the PCFG changed in such a way that the new costs of these two programs are $\text{cost}(\texttt{0}) = 10$ and $\text{cost}(\texttt{(indexof arg "+")}) = 7$. Let us examine different options for the subsequent synthesis cycle.

(1) *Restart from scratch:* If we restart the search with an empty bank, the program `(indexof arg "+")` is now encountered before the program `0` and selected as the representative of it equivalence class. In other words, the desired the behavior under the new PCFG is that the class $[0,0,0]$ has cost 7. Can we achieve this behavior without restarting the search?

(2) *Keep the bank unchanged:* Resuming the enumeration with B unchanged would be incorrect: in this case the representative of $[0,0,0]$ is still the program `0` with cost 5. As a result, any program we build in the new cycle that uses this equivalence class as a sub-program would have a wrong cost, and hence the enumeration order would be different from that prescribed by the new PCFG.

(3) *Re-index the bank:* Another option is to keep the programs stored in B but re-index it with their updated costs: for example, index the program `0` with cost 10. This does not solve the problem, however: now class $[0,0,0]$ has cost 10 instead the desired cost 7, because it still has a wrong representative in B. Therefore, in order to enforce the correct enumeration order in the new cycle we need to update the equivalence class representatives stored in the bank.

(4) *Update representatives:* To be able to update the representatives, we need to store the redundant programs in the bank instead of discarding them. To this end, prior work [131, 169, 168] has proposed representing the bank as a *finite tree automaton*, *i.e.* a hypergraph where nodes

correspond to equivalence classes (such as $[0,0,0]$) and edges correspond to productions (with the corresponding arity). The representative program of an equivalence class can be computed as the shortest hyper-path to the corresponding node from the set of initial nodes (inputs and literals); the cost of the class is the length of such a shortest path. When the PCFG is updated, leading to modified costs of hyper-edges, shortest paths for all nodes in this graph need to be recomputed. Algorithms for doing so [66] have super-linear complexity in the number of affected nodes. Since in our case most nodes are likely to be affected by the update, and since the number of nodes in the hypergraph is the same as the size of our bank B, this update step is roughly as expensive as rebuilding the bank from scratch. In addition, for a search space as large as the one PROBE explores for the SYGUS String benchmarks, the memory overhead of storing the entire hypergraph is also prohibitive.

Since restarting the search is expensive, PROBE does not return from the guided search immediately once a partial solution is found and instead keeps searching until a fixed cost limit and returns partial solutions in batches. There is a trade-off between restarting synthesis too often (wasting time exploring small programs again and again) and restarting too infrequently (wasting time on unpromising parts of the search space when an updated PCFG could guide the search better). In our implementation, we found that setting the cost limit to $6 \cdot C$ works best empirically, where C is the maximum production cost in the initial PCFG (this roughly corresponds to enumerating programs in size increments of six with the initial grammar).

3.6 Experiments

We have implemented the PROBE synthesis algorithm in Scala[8]. In this section, we empirically evaluate how PROBE compares to the baseline and state-of-the-art synthesis techniques. We design our experiments to answer the following research questions:

(**Q1**) How effective is the just-in-time learning in PROBE? We examine this question in two

[8] https://github.com/shraddhabarke/probe.git

parts:

1. by comparing PROBE to unguided bottom-up enumerative techniques, and
2. by comparing different schemes for partial solution selection.

(**Q2**) Is PROBE faster than state-of-the-art SYGUS solvers?

(**Q3**) Is the quality of PROBE solutions comparable with state-of-the-art SYGUS solvers?

3.6.1 Experimental Setup

We evaluate PROBE on three different application domains: string (STRING), bit-vector manipulation (BITVEC), and circuit transformations (CIRCUIT). We perform our experiments on a set of total 140 benchmarks, 82 of which are STRING benchmarks, 27 are BITVEC benchmarks and 31 are CIRCUIT benchmarks. The grammars containing the available operations for each of these domains are in Fig. 3.13 , Fig. 3.14, and Fig. 3.15.

STRING *Benchmarks.* The 82 STRING benchmarks are taken from the testing set of EUPHONY [3]. The entire EUPHONY String benchmark suite consists of 205 problems, from the PBE-String track of the 2017 SYGUS competition and from string-manipulation questions from popular online forums. EUPHONY uses 82 out of these 205 benchmarks as their testing set based on the criterion that EUSOLVER [13] could not solve them within 10 minutes. STRING benchmark grammars have a median of 16 operations, 11 literals, and 1 variable. All these benchmarks use input-output examples as semantic specification, and the number of examples ranges from 2 to 400.

BITVEC *Benchmarks.* The 27 BITVEC benchmarks originate from the book *Hacker's Delight* [170], commonly referred to as the bible of bit-twiddling hacks. We took 20 of them verbatim from the SYGUS competition suite: these are all the highest difficulty level (d5) Hacker's Delight benchmarks in SYGUS. We then found 7 additional loop-free benchmarks in synthesis literature [90, 77] and manually encoded them in the SYGUS format. BITVEC

benchmark grammars have a median of 17 operations, 3 literals, and 1 variable. The semantic specification of BITVEC benchmarks is a universally-quantified first-order formula that is functionally equivalent to the target program.

Note that in addition to Hacker's Delight benchmarks, the SYGUS bitvector benchmark set also contains EUPHONY bitvector benchmarks. We decided to exclude these benchmarks from our evaluation because they have very peculiar solutions: they all require extensive case-splitting, and hence are particularly suited to synthesizers that perform *condition abduction* [13, 96, 5]. Since PROBE (unlike EUPHONY) does not implement condition abduction, it is bound to perform poorly on these benchmarks. At the same time, condition abduction is orthogonal to the techniques introduced in this paper; hence PROBE's performance on these benchmarks would not be informative.

CIRCUIT *Benchmarks.* The 31 CIRCUIT benchmarks are taken from the EUPHONY testing set. These benchmarks involve synthesizing constant-time circuits that are cryptographically resilient to timing attacks. CIRCUIT benchmark grammars have a median of 4 operations, 0 literals, and 6 variables. The semantic specification is a universally-quantified boolean formula functionally equivalent to the circuit to be synthesized.

Reducing First-order Specifications to Examples. As discussed above, only the string domain uses input-output examples as the semantic specification, while the other two domains use a more general SYGUS formulation where the specification is a (universally-quantified) first-order formula. We extend PROBE to handle the latter kind of specifications in a standard way (see *e.g.* [13]), using *counter-example guided inductive synthesis* (CEGIS) [152]. CEGIS proceeds in iterations, where each iteration first *synthesizes* a candidate program that works on a finite set of inputs, and then *verifies* this candidate against the full specification, adding any failing inputs to the set of inputs to be considered in the next synthesis iteration. We use PROBE for the synthesis phase of the CEGIS loop. At the start of each CEGIS iteration, we initialize an independent instance of PROBE starting from a uniform grammar.

Baseline Solvers. As the state-of-the-art in research questions (**Q2**) and (**Q3**) we use EUPHONY and CVC4, which are the state-of-the-art SYGUS solvers in terms of performance and solution quality. EUPHONY [102] also uses probabilistic models to guide its search, but unlike PROBE they are pre-learned models. We used the trained models that are available in EUPHONY's repository [3]. CVC4 [136] has been the winner of the PBE-Strings track of the SYGUS Competition [12] since 2017. We use the CVC4 version 1.8 (Aug 6 2020 build).

Start →	*S*	
S →	`arg0` | `arg1` | ...	string variables
|	`lit-1` | `lit-2` | ...	string literals
|	(`replace` *S S S*)	`replace s x y` replaces first occurrence of `x` in `s` with `y`
|	(`concat` *S S*)	`concat x y` concatenates `x` and `y`
|	(`substr` *S I I*)	`substr x y z` extracts substring of length `z`, from index `y`
|	(`ite` *B S S*)	`ite x y z` returns `y` if `x` is true, otherwise `z`
|	(`int.to.str` *I*)	`int.to.str x` converts int `x` to a string
|	(`at` *S I*)	`at x y` returns the character at index `y` in string `x`
B →	`true` | `false`	bool literals
|	(= *I I*)	`= x y` returns true if `x` equals `y`
|	(`contains` *S S*)	`contains x y` returns true if `x` contains `y`
|	(`suffixof` *S S*)	`suffixof x y` returns true if `x` is the suffix of `y`
|	(`prefixof` *S S*)	`prefixof x y` returns true if `x` is the prefix of `y`
I →	`arg0` | `arg1` | ...	int variables
|	`lit-1` | `lit-2` | ...	int literals
|	(`str.to.int` *S*)	`str.to.int x` converts string `x` to a int
|	(+ *I I*)	`+ x y` sums `x` and `y`
|	(- *I I*)	`- x y` subtracts `y` from `x`
|	(`length` *S*)	`length x` returns length of `x`
|	(`ite` *B I I*)	`ite x y z` returns `y` if `x` is true, otherwise `z`
|	(`indexof` *S S I*)	`indexof x y z` returns index of `y` in `x`, starting at index `z`

Figure 3.13: The full SYGUS STRING grammar of the EUPHONY benchmark suite. Integer and string variables and constants change per benchmark. Some benchmark files contain a reduced grammar.

Start →	*BV*	
BV →	`arg0` | `arg1` | ...	bit-vector variables
|	`lit-1` | `lit-2` | ...	bit-vector literals
|	(`xor` *BV BV*)	`xor x y` performs bitwise xor between `x` and `y`
|	(`and` *BV BV*)	`and x y` performs bitwise and operation between `x` and `y`
|	(`or` *BV BV*)	`or x y` performs bitwise or operation between `x` and `y`
|	(`neg` *BV*)	`neg x` returns the two's complement of `x`
|	(`not` *BV*)	`not x` returns the one's complement of `x`
|	(`add` *BV BV*)	`add x y` adds `x` and `y`
|	(`mul` *BV BV*)	`mul x y` multiplies `x` and `y`
|	(`udiv` *BV BV*)	`udiv x y` returns the unsigned quotient of dividing `x` by `y`
|	(`urem` *BV BV*)	`urem x y` returns the unsigned remainder of dividing `x` by `y`
|	(`lshr` *BV BV*)	`lshr x y` returns the logical right shift of `x` by `y` bits
|	(`ashr` *BV BV*)	`ashr x y` returns the arithmetic right shift of `x` by `y`
|	(`shl` *BV BV*)	`shl x y` returns the logical left shift of `x` by `y`
|	(`sdiv` *BV BV*)	`sdiv x y` returns the signed quotient of dividing `x` by `y`
|	(`srem` *BV BV*)	`srem x y` returns the signed remainder of dividing `x` by `y`
|	(`sub` *BV BV*)	`sub x y` subtracts `y` from `x`
|	(`ite` *B BV BV*)	`ite x y z` returns `y` if `x` is true, otherwise `z`
B →	`true` | `false`	bool literals
|	(`=` *BV BV*)	`= x y` returns true if `x` equals `y`
|	(`ult` *BV BV*)	`ult x y` returns true if `x` is unsigned less than `y`
|	(`ule` *BV BV*)	`ule x y` returns true if `x` is unsigned less than equal to `y`
|	(`slt` *BV BV*)	`slt x y` returns true if `x` is signed less than `y`
|	(`sle` *BV BV*)	`sle x y` returns true if `x` is signed less than equal to `y`
|	(`ugt` *BV BV*)	`ugt x y` returns true if `x` unsigned greater than `y`
|	(`redor` *BV*)	`redor x` performs bit-wise or reduction of `x`
|	(`and` *BV BV*)	`and x y` returns the logical and of `x` and `y`
|	(`or` *BV BV*)	`or x y` returns the logical or of `x` and `y`
|	(`not` *BV*)	`not x` returns the logical not of `x`

Figure 3.14: The full SyGuS BitVec grammar of the Hacker's Delight benchmarks; variables and constants change per benchmark. Some of the benchmarks contain a reduced grammar; required constants are provided.

Start →	*B*	
B →	`arg0` | `arg1` | ...	boolean variables
|	(`and` *B B*)	`and x y` returns the logical and of `x` and `y`
|	(`not` *B*)	`not x` returns the logical not of `x`
|	(`or` *B B*)	`or x y` returns the logical or of `x` and `y`
|	(`xor` *B B*)	`xor x y` returns the logical xor of `x` and `y`

Figure 3.15: The full SYGUS CIRCUIT grammar of the EUPHONY benchmark suite. Variables and the depth of the grammar change per benchmark.

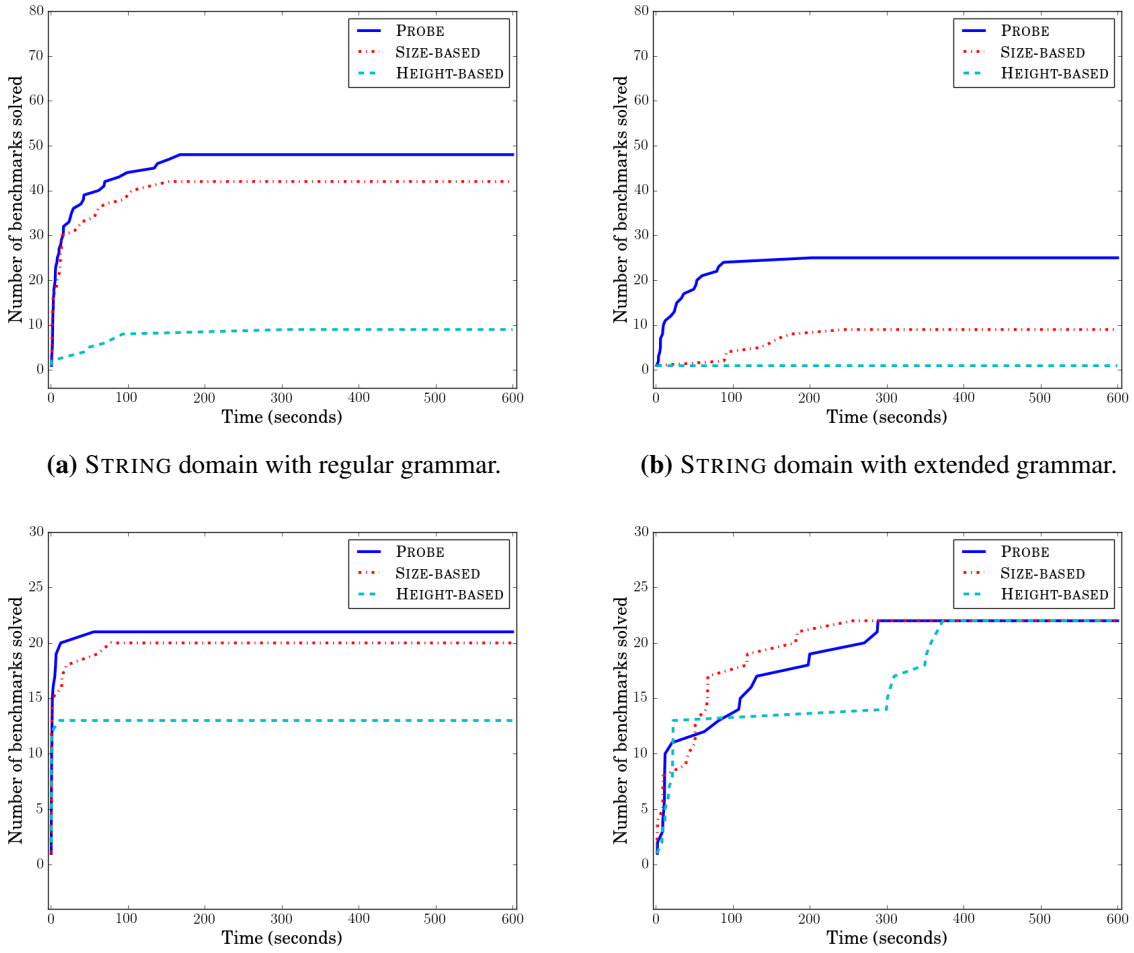
Experimental Setup. All experiments were run with a 10 minute timeout for all solvers, on a commodity Lenovo laptop with a i7 quad-core CPU @ 1.90GHz with 16GB of RAM.

3.6.2 Q1.1: Effectiveness of Just-in-time Learning

To assess the effectiveness of the just-in-time learning approach implemented in PROBE, we first compare it to two unguided bottom-up search algorithms: height-based and size-based enumeration. We implement these baselines inside PROBE, as simplifications of guided bottom-up search.

Results for* STRING *Domain. We measure the time to solution for each of the 82 benchmarks in the STRING benchmark set, for each of the three methods: PROBE, size-based, and height-based enumeration. The results are shown in Fig. 3.16a. PROBE, size-based and height-based enumeration are able to solve 48, 42 and 9 problems, respectively. Additionally, at every point after one second, PROBE has solved more benchmarks than either size-based or height-based enumeration.

Just-in-time Learning and Grammar Size. In addition to our regular benchmark suite, we created a version of the STRING benchmarks (except 12 outliers that have abnormally many string literals) that uses an *extended string grammar*, which includes all operations and literals from all STRING benchmarks. In total this grammar has all available string, integer and boolean



(a) STRING domain with regular grammar.

(b) STRING domain with extended grammar.

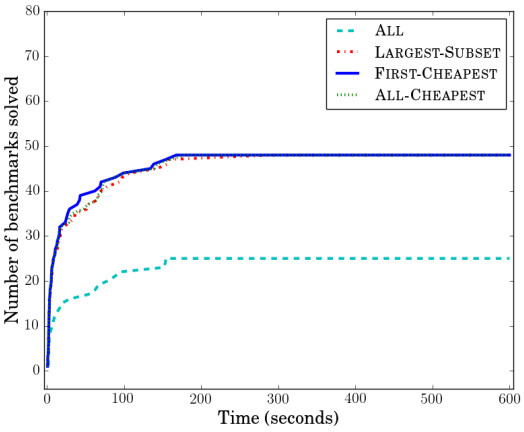
(c) BITVEC domain

(d) CIRCUIT domain

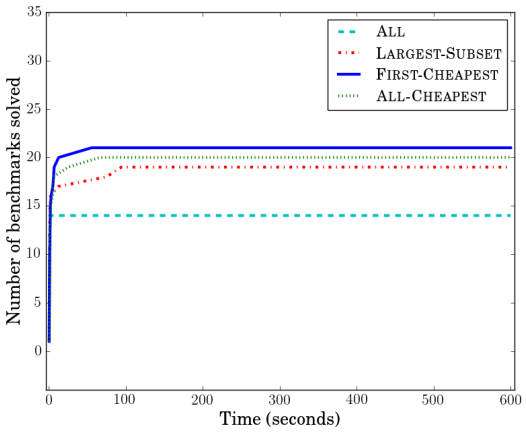
Figure 3.16: Number of benchmarks solved by PROBE and unguided search techniques (size-based and height-based enumeration) for STRING, BITVEC and CIRCUIT domains. Timeout is 10 min, graph scale is linear.

operations in the SYGUS language specification and 48 string literals and 11 integer literals. These 70 extended-grammar benchmarks allow us to test the behavior of PROBE on larger grammars and thereby larger program spaces.

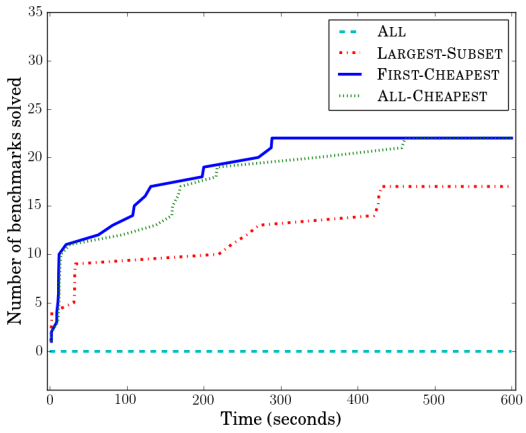
Within a timeout of 10 minutes, PROBE solves 25 benchmarks (52% of the original number) whereas height-based and size-based enumeration solved 1 (11% of original) and 9 (21% of original) benchmarks respectively as shown in Fig. 3.16b. We find this particularly encouraging, because the size of the grammar usually has a severe effect on the synthesizer (as



(a) STRING domain

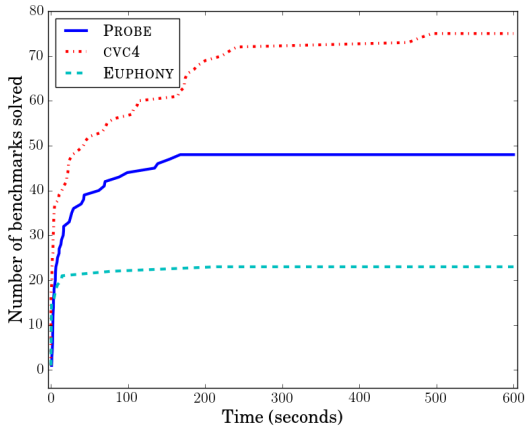


(b) BITVEC domain

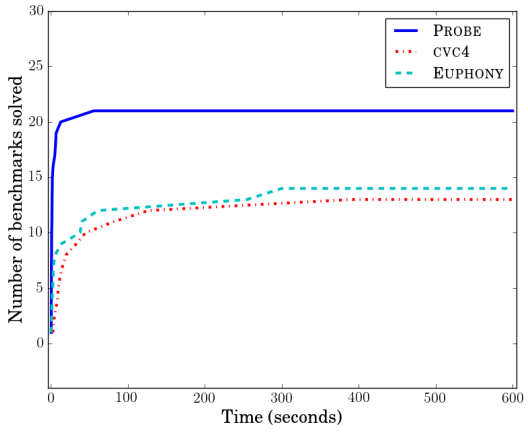


(c) CIRCUIT domain

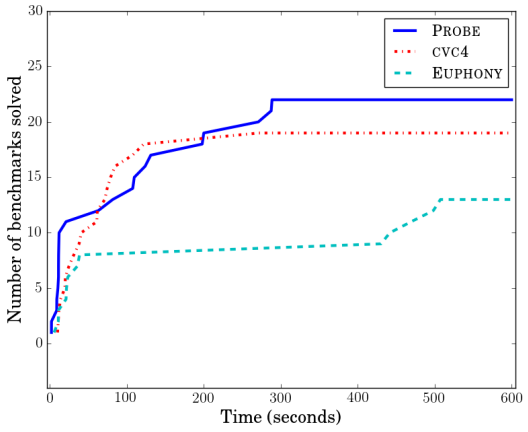
Figure 3.17: Number of benchmarks solved by PROBE with schemes for selecting promising partial solutions. Schemes are described in Sec. 3.5.2; ALL represents no selection (all partial solutions are used to update the PCFG). Timeout is 10 min, graph scale is linear.



(a) STRING domain



(b) BITVEC domain



(c) CIRCUIT domain

Figure 3.18: Number of benchmarks solved by PROBE, EUPHONY and CVC4 for STRING, BITVEC and CIRCUIT domains. Timeout is 10 min, graph scale is linear.

we can see for size-based enumeration), so much so that carefully constructing a grammar is considered to be part of synthesizer design. While the baseline synthesizers need the benefit of approaching each task with a different, carefully chosen grammar, PROBE's just-in-time learning is much more robust to additional useless grammar productions. Even with a larger grammar, PROBE's search space does not grow as much: once it finds a partial solution, it hones in on the useful parts of the grammar.

Results for* BITVEC *Domain. The results for the BITVEC benchmarks are shown in Fig. 3.16c. Out of the 27 BITVEC benchmarks, PROBE, size-based and height-based solve 21,

20 and 13 benchmarks, respectively. In addition to solving one more benchmark, PROBE is also considerably faster than size-based enumeration, as we can see from the horizontal distance between the two curves on the graph. PROBE significantly outperforms the baseline height-based enumeration technique.

Results for* CIRCUIT *Domain. The results for the CIRCUIT benchmarks are shown in Fig. 3.16d. Each of the three techniques solves 22 out of 31 benchmarks, with size-based enumeration outperforming PROBE in terms of synthesis times. The reason PROBE performs worse in this domain is that the CIRCUIT grammar is very small (only four operations in the median case) and the solutions tend to use most of productions from the grammar. Thus, rewarding specific productions in the PCFG does not yield significant benefits, but in fact the search is slowed down due to the restarting overhead incurred by PROBE.

Summary of Results. Out of the 210 benchmarks from three different domains and the extended STRING grammar, PROBE solves 116, size-based solves 93 and height-based solves 45. We conclude that overall, **PROBE outperforms both baseline techniques, and is therefore an effective synthesis technique.**

3.6.3 Q1.2: Selection of Partial Solutions

In this section, we empirically evaluate the schemes for selecting promising partial solutions. We compare four different schemes: the three described in Sec. 3.5.2 and the baseline of using ALL generated partial solutions. The results are shown in Fig. 3.17.

The ALL baseline scheme performs consistently worse than the other schemes on all three domains (and also worse than unguided size-based enumeration). For the circuit domain (Fig. 3.17c), the ALL scheme solves none of the benchmarks. The performance of the remaining schemes is very similar, indicating that the general idea of leveraging small and semantically unique partial solutions to guide search is robust to minor changes in the selection criteria. We select FIRST CHEAPEST as the scheme used in PROBE since it provides a balance between rewarding few partial solutions while still considering syntactically different approaches.

3.6.4 Q2: Is PROBE Faster than the State-of-the-art?

We compare PROBE's time to solution on the benchmarks in our suite against two state-of-the-art SYGUS solvers, EUPHONY and CVC4. The results for all three domains are shown in Fig. 3.18.

STRING *Domain.* Results for the STRING domain are shown in Fig. 3.18a. Of the 82 benchmarks in the STRING suite, PROBE solves 48 benchmarks, with an average time of 29s and a median time of 8.3s. EUPHONY solves 23 benchmarks, with average of 15.4s and a median of 0.7s. CVC4 solves 75 benchmarks, with an average of 61.8s and a median of 10.2s.

The performance of EUPHONY is close to that reported originally by [102]; they report 27 of the 82 benchmarks solved with a 60 minute timeout. Even with the reduced timeout, PROBE vastly outperforms EUPHONY.

When only examining time to solution, CVC4 outperforms PROBE: not only does it solve more benchmarks faster, but it still solves new benchmarks long after PROBE and EUPHONY have plateaued. However, these solutions are not necessarily usable, as we show in Sec. 3.6.5.

BITVEC *Domain.* Out of the 27 BITVEC benchmarks, PROBE solves 21 benchmarks, EUPHONY solves 14 and CVC4 solves 13 benchmarks as shown in Fig. 3.18b. PROBE outperforms both CVC4 and EUPHONY on these benchmarks with an average time of 5s and median time of 1.5s. EUPHONY's average time is 52s and median is 4.6s while CVC4 takes an average of 58s and a median of 15s. PROBE not only solves the most benchmarks overall, it also solves the highest number of benchmarks compared to EUPHONY and CVC4 at each point in time.

We should note that the EUPHONY model we used for this experiment was trained on the EUPHONY set of bit-vector benchmarks (the ones we excluded because of the case-splits) rather than the Hacker's Delight benchmarks. Although EUPHONY does very well on its own bit-vector benchmarks, it does not fare so well on Hacker's Delight. These results shed some light on how brittle pre-trained models are in the face of subtle changes in syntactic program properties, even within a single bit-vector domain; we believe this makes a case for just-in-time learning.

CIRCUIT *Domain.* Out of the 31 CIRCUIT benchmarks, PROBE solves 22 benchmarks with an average time of 90s and median time of 42s (see Fig. 3.18c). EUPHONY solves 13 benchmarks with average and median times of 193.6s and 36s. CVC4 solves 19 benchmarks with average and median times of 60s and 41s. PROBE outperforms both CVC4 and EUPHONY in terms of the number of benchmarks solved. Moreover CVC4 generates much larger solutions than PROBE, as discussed in Sec. 3.6.5.

Summary of Results. Of the total 140 benchmarks, PROBE solves 91 within the 10-minute timeout, EUPHONY solves 50, and CVC4 solves 107. PROBE outperforms EUPHONY's pre-learned models in all three domains, and while CVC4 outperforms PROBE in the STRING domain; the next subsection will discuss the quality of the results it generates.

3.6.5 Q3: Quality of Synthesized Solutions

So far, we have tested the ability of solvers to arrive at *a* solution, without checking what the solution is. When a PBE synthesizer finds a program for a given set of examples, it guarantees nothing but the behavior on those examples. Indeed, the SYGUS Competition scoring system[9] awards the most points (five) for simply returning any program that matches the given examples. It is therefore useful to examine the *quality* of the solutions generated by PROBE and its competition.

Size is a common surrogate measure for program *simplicity*: e.g., the SYGUS Competition awards an additional point to the solver that returns the smallest program for each benchmark. Program size reflects two sources of complexity: (i) unnecessary operations that do not influence the result, and, perhaps more importantly, (ii) *case splitting* that overfits to the examples. It is therefore reasonable to assume that a smaller solution is more interpretable and generalizes better to additional inputs beyond the initial input-output examples.

Based on these observations, we first estimate the quality of results for all three domains by comparing the sizes of solutions generated by PROBE and other tools. We next focus on the

[9] https://sygus.org/comp/2019/results-slides.pdf, slide 13

STRING benchmarks, as this is the only domain where the specification is given in the form of input-output examples, and hence is prone to overfitting. For this domain, we additionally measure the number of case splits in generated solutions and test their generalization accuracy on unseen inputs.

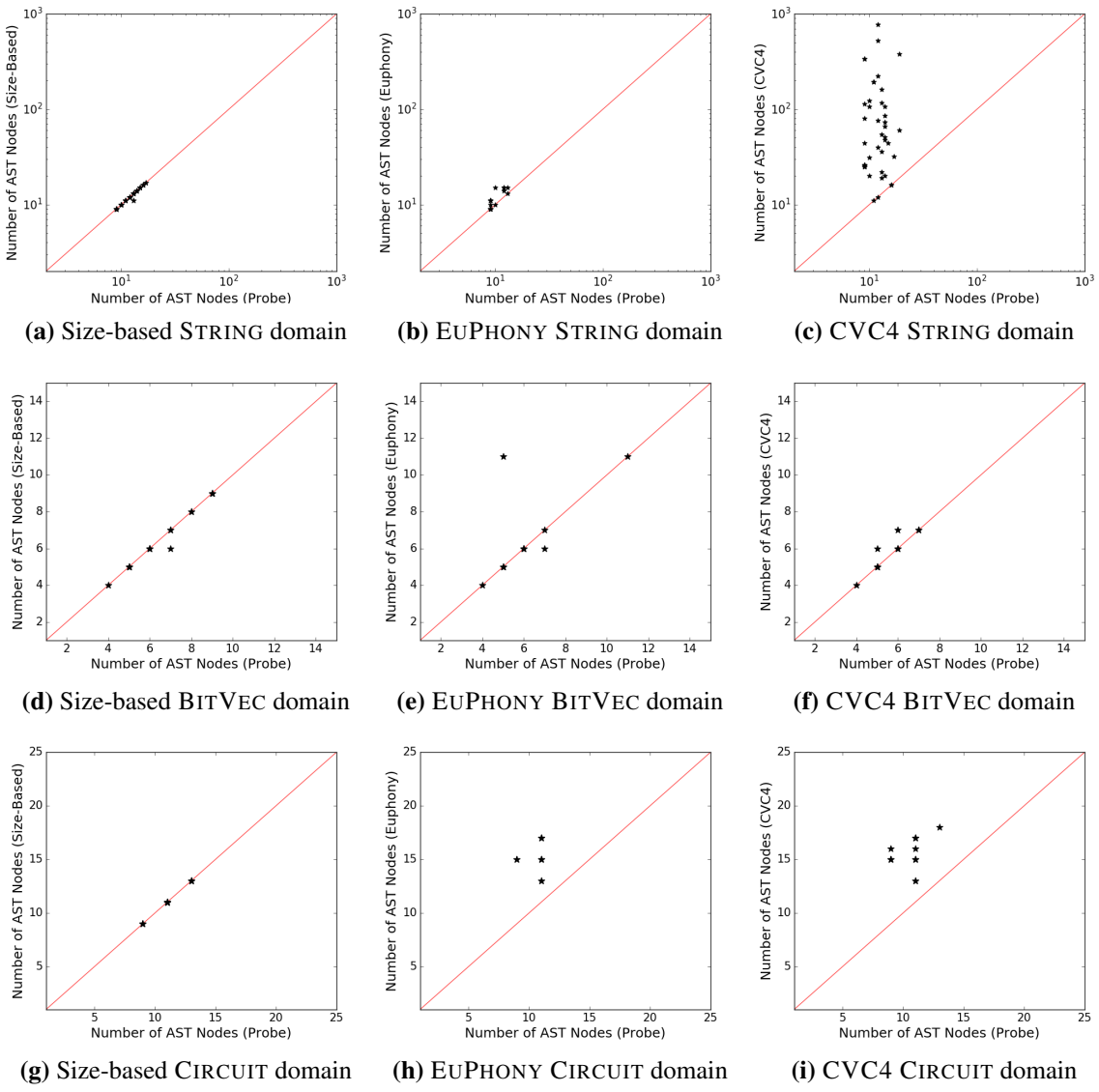
Size of Generated Solutions. Fig. 3.19 shows the sizes of PROBE solutions in AST nodes, as compared to size-based enumeration (which always returns the smallest solution by definition), as well as EUPHONY and CVC4. Each comparison is limited to the benchmarks both tools can solve.

STRING *Domain.* First, we notice in Fig. 3.19a that PROBE sometimes finds larger solutions than size-based enumeration, but the difference is small. Likewise, Fig. 3.19b shows that EUPHONY and PROBE return similar-sized solutions. PROBE returns the smaller solutions for 10 benchmarks, but the difference is not large. On the other hand, CVC4 solutions (Fig. 3.19c) are larger than PROBE's on 41 out of 45 benchmarks, sometimes by as much as *two orders of magnitude*. For the remaining four benchmarks, solution sizes are equal. On one of the benchmarks not solved by PROBE (and therefore not in the graph), CVC4 returns a result with over 7100(!) AST nodes.

Other Domains. Fig. 3.19d shows that on the BITVEC domain PROBE finds the minimal solution in all cases except one. Solutions by EUPHONY (Fig. 3.19e) and CVC4 (Fig. 3.19f) are slightly larger[10] in one (resp. two) cases, but the difference is small. For the CIRCUIT benchmarks, PROBE always finds minimal solutions, as shown in Fig. 3.19g. Both EUPHONY (Fig. 3.19h) and CVC4 (Fig. 3.19i) generate larger solutions for *all* of the commonly solved benchmarks. Hence, on the CIRCUIT domain, PROBE outperforms its competitors with respect to *both* synthesis time and solution size.

Case Splitting. So why are the CVC4 STRING programs so large? Upon closer examination, we determined that they perform over-abundant *case splitting*, which hurts both readability

[10]Note that we use linear scale for BITVEC and CIRCUIT as opposed to logarithmic scale for STRING.

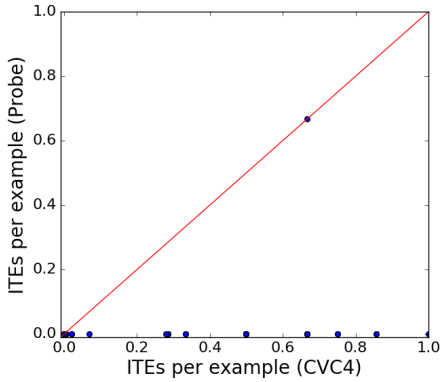


(a) Size-based STRING domain **(b)** EUPHONY STRING domain **(c)** CVC4 STRING domain

(d) Size-based BITVEC domain **(e)** EUPHONY BITVEC domain **(f)** CVC4 BITVEC domain

(g) Size-based CIRCUIT domain **(h)** EUPHONY CIRCUIT domain **(i)** CVC4 CIRCUIT domain

Figure 3.19: Comparison between sizes of programs generated by different algorithms. Fig. 3.19a, Fig. 3.19b and Fig. 3.19c compare PROBE *vs.* size-based enumeration, EUPHONY and CVC4, respectively, on the STRING domain; graphs are log scale. Fig. 3.19d, Fig. 3.19e and Fig. 3.19f compare the same pairs of tools on the BITVEC domain and Fig. 3.19g, Fig. 3.19h and Fig. 3.19i on the CIRCUIT domain; graphs are linear scale.



(a) Number of `ite` operations per examples

Test Benchmark	Training Examples	Testing Examples	PROBE Accuracy	CVC4 Accuracy
initials-long	4	54	100%	100%
phone-5-long	7	100	100%	100%
phone-6-long	7	100	100%	100%
phone-7-long	7	100	100%	7%
phone-10-long	7	100	100%	57%
phone-9-long	7	100	N/A	7%
univ_4-long	8	20	N/A	73.6%
univ_5-long	8	20	N/A	68.4%
univ_6-long	8	20	N/A	100%
Avg Accuracy			100%	68.1%

(b) Generalization accuracy on unseen inputs

Figure 3.20: Fig. 3.20a displays the number of `ite` operations per example for the STRING benchmarks solved by PROBE and CVC4. CVC4 has a large number of case splits as indicated. Fig. 3.20b shows the generalization accuracy on unseen inputs for the 9 test benchmarks.

Benchmark	Solution generated	Time (s)
stackoverflow1.sl	`(substr arg 0 (+ (indexof arg "Inc" 1) -1))`	2.2s
stackoverflow3.sl	`(substr arg (- (to.int (concat "1" "9")) 2) (len arg))`	2.1s
stackoverflow8.sl	`(substr arg (- (len arg) (+ (+ 2 4) 4)) (len arg))`	6.5s
stackoverflow10.sl	`(substr arg (indexof (replace arg " " (to.str (len arg))) " " 1) 4)`	27.6s
exceljet1.sl	`(substr arg1 (+ (indexof arg1 "_" 1) 1) (len arg1))`	1.5s
exceljet2.sl	`(replace (substr arg (- (len arg) (indexof arg "." 1)) (len arg)) "." "")`	16.5s
initials.sl	`(concat (concat (at name 0) ".") (concat (at name (+ (indexof name " " 0) 1)) "."))`	134.5s
phone-6-long.sl	`(substr name (- (indexof name "-" 4) 3) 3)`	3.4s
43606446.sl	`(substr arg (- (len arg) (+ (+ 1 1) (+ 1 1))) (+ (+ 1 1) 1))`	10.8s
11604909.sl	`(substr (concat " " arg) (indexof arg "." 1) (+ (+ 1 1) 1))`	15.9s

Figure 3.21: PROBE solutions for 10 randomly selected benchmarks out of the 48 benchmarks PROBE solves from the [3] STRING testing set, Time indicates the synthesis time in seconds.

and generality. To confirm our intuition, we count the number of if-then-else operations (`ite`) in the programs synthesized by PROBE and by CVC4. The results are plotted in Fig. 3.20a. The number of `ites` is normalized by number of examples in the task specification. PROBE averages 0.01 `ite` per example (for all but one benchmark PROBE solutions do not contain an `ite`), whereas CVC4 averages 0.42 `ites` per example. When also considering benchmarks PROBE cannot solve, some CVC4 programs have more than two `ites` per example.

Generalization Accuracy. Finally, we test the generality of the synthesized programs—whether they generalize well to additional examples, or in other words, whether synthesis returns reusable code. Concretely, we measure *generalization accuracy* [8], the percentage of unseen inputs for which a generated program produces the correct output. To this end, we find a solution

Benchmark	Solution generated	Time (s)
hd-11.sl	`(bvult y (bvand x (bvnot y)))`	2.4s
hd-09.sl	`(bvsub x (bvshl (bvand (bvashr x #x000000000000001f) x) #x0000000000000001))`	6.3s
hd-15.sl	`(bvsub (bvor x y) (bvlshr (bvxor x y) #x0000000000000001))`	13s
hd-18.sl	`(bvult (bvxor x (bvneg x)) (bvneg x))`	1.3s
hd-13.sl	`(bvor (bvashr x #x000000000000001f) (bvlshr (bvneg x) #x000000000000001f))`	1.6s

Figure 3.22: PROBE solutions for 5 randomly selected benchmarks out of the 21 benchmarks PROBE solves from the Hacker's Delight BITVEC set, Time indicates the synthesis time in seconds.

Benchmark	Solution generated	Time (s)
CrCy_10-sbox2-D5-sIn14.sl	`(xor LN200 (xor LN61 (and (xor LN16 LN17) LN4)))`	9.4s
CrCy_10-sbox2-D5-sIn88.sl	`(xor LN73 (and (xor (and LN70 (xor (xor LN236 LN252) LN253)) LN71) LN74))`	287.1s
CrCy_10-sbox2-D5-sIn78.sl	`(and (xor (and LN70 (xor (xor LN236 LN252) LN253)) LN73) LN77)`	11.8s
CrCy_10-sbox2-D5-sIn80.sl	`(xor LN73 (and LN70 (xor (xor LN236 LN252) LN253)))`	2.2s
CrCy_8-P12-D5-sIn1.sl	`(xor (xor (xor LN3 LN7) (xor (xor LN75 LN78) LN81)) k4)`	9.1s

Figure 3.23: PROBE solutions for 5 randomly selected benchmarks out of the 22 benchmarks PROBE solves from the [3] CIRCUIT set, Time indicates the synthesis time in seconds.

using PROBE and CVC4, and then test it on additional examples for the same program.

Since most benchmarks in our suite contain only a few input-output examples, splitting these examples into a training and testing set would render most benchmarks severely under-specified. Instead we turn to a subset of the STRING benchmarks from the SYGUS Competition PBE-Strings suite. These are benchmark pairs where each task appears in a "short" form with a small number of examples and a "long" form with additional examples, but both represent the same task and share the same grammar. There are nine such benchmark pairs in this suite.

We compare the generalization accuracy of CVC4 and PROBE by using the short benchmark of each pair to synthesize a solution, and, if a solution is found, we test it on the examples of the long version of the benchmark to see how well it generalizes. The results are shown in Fig. 3.20b.

The first part of the table shows the benchmarks where PROBE finds a solution. As discussed above, PROBE rarely finds solutions with case splits, so it is not surprising that once it finds a program, that program is not at all overfitted to the examples.

Solutions found by CVC4 generalize with 100% accuracy in 4 out of the 9 benchmark pairs. In two of the benchmarks, the accuracy of CVC4 solutions is only 7%, or precisely the

7 training examples out of the 100-example test set, representing a complete overfitting to the training examples. On average, CVC4 has 68% generalization accuracy on these benchmark pairs. Even though this experiment is small, it provides a glimpse into the extent to which CVC4 solutions sometimes overfit to the examples.

Sample Solutions. Finally, we examine a few sample solutions generated by PROBE in Fig. 3.21 for the STRING domain, Fig. 3.22 for the BITVEC domain and Fig. 3.23 for the CIRCUIT domain. Even though the SYGUS language is unfamiliar to most readers, we believe that these solutions should appear simple and clearly understandable. In comparison, the CVC4 solutions to these benchmarks are dozens or hundreds of operations long.

Solution Quality. The experiments in this section explored solution quality via three empirical measures: solution size, the number of case-splits, and the ability of solutions to generalize to new examples for the same task. These results show conclusively that, while CVC4 is considerably faster than PROBE, and solves more benchmarks, the quality of its solutions is significantly worse.

3.6.6 Conclusions

In conclusion, we have shown that PROBE is faster and solves more benchmarks than unguided enumerative techniques, which confirms that just-in-time learning is an improvement on a baseline synthesizer. We have also shown that PROBE is faster and solves more benchmarks than EUPHONY, a probabilistic synthesizer with a pre-learned model, based on top-down enumeration. Finally, we have explored the quality of synthesized solutions via size, case splitting, and generalizability, and found that even though CVC4 solves more benchmarks than PROBE, its solutions to example-based benchmarks overfit to the examples, and are therefore neither readable nor reusable; in contrast, PROBE's solutions are small and generalize perfectly.

3.7 Related Work

Enumerative Program Synthesis. Despite their simplicity, enumerative program synthesizers are known to be very effective: ESOLVER [9] and EUSOLVER [13] have been past winners of the SYGUS competition [12, 11]. Enumerative synthesizers typically explore the space of programs either top-down, by extending a partial program tree from the node towards the leaves [102, 13, 94, 98], or bottom-up, by gradually building up a program tree from the leaves towards the root [159, 5, 9, 127]. These two strategies have complementary strengths and weaknesses, similar to backward chaining and forward chaining in proof search.

One important advantage of bottom-up enumeration for inductive synthesis is the ability to prune the search space using *observational equivalence* (OE), *i.e.* discard a program that behaves equivalently to an already enumerated program on the set of inputs from the semantic specification. OE was first proposed in [159, 5] and since then has been successfully used in many bottom-up synthesizers [165, 127, 14], including PROBE. Top-down enumeration techniques cannot fully leverage OE, because incomplete programs they generate cannot be evaluated on the inputs. Instead, these synthesizers prune the space based on other syntactic and semantic notions of program equivalence: for example, [81, 124, 62] only produce programs in a normal form; [60, 96, 149] perform symmetry reduction based on equational theories (either built-in or user-provided); finally, EUPHONY [102] employs a weaker version of OE for incomplete programs, which compares their complete parts observationally and their incomplete parts syntactically.

Guiding Synthesis with Probabilistic Models. Recent years have seen proliferation of probabilistic models of programs [7], which can be used, in particular, to guide program synthesis. The general idea is to prioritize the exploration of grammar productions based on scores assigned by a probabilistic model; the specific technique, however, varies depending on (1) the context taken into consideration by the model when assigning scores, and (2) how the scores are taken into account during search. Like PROBE, [98, 19, 116] use a PCFG, which assigns scores to productions *independently of their context* within the synthesized program;

unlike PROBE, however, these techniques select the PCFG once, at the beginning of the synthesis process, based on a learned mapping from semantic specifications to scores. On the opposite end of the spectrum, METAL [147] and CONCORD [35] use graph-based and sequence-based models, respectively, to condition the scores on the *entire partial program* that is being extended. In between these extremes, EUPHONY uses a learned context in the form of a *probabilistic higher-order grammar* [23], while NGDS [94] conditions the scores on the *local specification* propagated top-down by the deductive synthesizer. The more context a model takes into account, the more precise the guidance it provides, but also the harder it is to learn. Another consideration is that neural models, used in [94, 147, 35] incur a larger overhead than simple grammar-based models, used in PROBE and [116, 19, 98, 102], since they have to invoke a neural network at each branching point during search.

As for using the scores to guide search, most existing techniques are specific to *top-down enumeration*. They include prioritized depth-first search [19], branch and bound search [94], and variants of best-first search [116, 102, 98]. In contrast to these approaches, PROBE uses the scores to guide *bottom-up enumeration* with observational equivalence reduction. PROBE's enumeration is essentially a bottom-up version of best-first search, and it empirically performs better than the top-down best-first search in EUPHONY; one limitation, however, is that our algorithm is specific to PCFGs and extending it to models that require more context is not straightforward.

DEEPCODER [19] also proposes a scheme they call *sort and add*, which is not specific to top-down enumeration and can be used in conjunction with any synthesis algorithm: this scheme runs synthesis with a reduced grammar, containing only productions with highest scores, and iteratively adds less likely productions if no solution is found. Although very general, this scheme is less efficient than best-first search: it can waste resources searching with an insufficient grammar, and has to revisit the same programs again once the search is restarted with a larger grammar.

Finally, METAL and CONCORD, which are based on reinforcement learning (RL), do

not perform traditional backtracking search at all. Instead, at each branching point, they simply choose a single production that has the highest score according to the current RL policy; a sequence of such decisions is called a *policy rollout*. If a rollout does not lead to a solution, the policy is updated according to a reward function explained below and a new rollout is performed from scratch.

Learning Probabilistic Models. Approaches to *learning* probabilistic models of programs can be classified into two categories: pre-training and learning on the fly. In the first category, [116], EUPHONY, and NGDS are trained using a large corpus of human-designed synthesis problems and their gold standard solutions (the latter can be provided by a human or synthesized using size-based enumeration). Such datasets are costly to obtain: because these models are domain-specific, a new training corpus has to be designed for each domain. In contrast, DEEPCODER learns from randomly sampled programs and inputs; it is, however, unclear how effective this technique is for domains beyond the highly restricted DSL in the paper. Unlike all these approaches, PROBE requires no pre-training, and hence can be used on a new domain without any up-front cost; if a pre-trained PCFG for the domain is available, however, PROBE can also be initialized with this model (although we have not explored this avenue in the present work).

DREAMCODER, METAL, and CONCORD are related to the just-in-time approach of PROBE in the sense that they update their probabilistic model on the fly. DREAMCODER learns a probabilistic model from *full solutions* to a subset of synthesis problems from a corpus, whereas PROBE learns a problem-specific model from *partial solutions* to a single synthesis problem.

The RL-based tools METAL and CONCORD start with a pre-trained RL policy and then fine-tune it for the specific task during synthesis. Note that off-line training is vital for the performance of these tools, while PROBE is effective even without a pre-trained model. The *reward mechanism* in METAL is similar to PROBE: it rewards a policy based on the fraction of input-output examples solved by its rollout. CONCORD instead rewards its policies based

on infeasibility information from a deductive reasoning engine: productions that expand to infeasible programs have lower probability in the next rollout. Although the CONCORD paper reports that its reward mechanism outperforms that of METAL, we conjecture that rewards based on partial solutions are simply not as good a fit for RL as they are for bottom-up enumeration: as we discuss in Sec. 3.5.2, it is crucial to learn from *shortest* partial solutions to avoid irrelevant syntactic features; policy rollouts do not guarantee that short solutions are generated first. Finally, CONCORD's reward mechanism requires expensive solver invocations to check infeasibility of partial programs, while PROBE's reward mechanism incurs practically no overhead compared to unguided search.

Leveraging Partial Solutions to Guide Synthesis. LASY [130] and FRANGEL [146] are component-based synthesis techniques that leverage information from partial solutions to generate new programs. LASY explicitly requires the user to arrange input-output examples in the order of increasing difficulty, and then synthesizes a sequence of programs, where i^{th} program passes the first i examples. Each following program is not synthesized from scratch, but rather by modifying the previous program; hence intermediate programs serve as "stepping stones" for synthesis. PROBE puts less burden on the user: it does not require the examples to be arranged in a sequence, and instead identifies partial solutions that satisfy any subset of examples.

Similar to PROBE, FRANGEL leverages partial solutions that satisfy any subset of the example specification. FRANGEL generates new programs by randomly combining fragments from partial solutions. PROBE is similar to FRANGEL and LASY in that it guides the search using syntactic information learned from partial solutions, but we achieve that by updating the weights of useful productions in a probabilistic grammar and using it to guide bottom-up enumerative search.

Our previous work, BESTER [127] proposes a technique to accumulate multiple partial solutions during bottom-up enumerative synthesis with minimum overhead. PROBE is a natural extension of BESTER: it leverages these accumulated partial solutions to guide search.

During top-down enumeration, [98] employs an optimization strategy where the cost of an incomplete (partial) program is lowered if it satisfies some of the examples. This optimization encourages the search to complete a partial program that looks promising, but unlike PROBE, offers no guidance on which are the likely productions to complete it with. Moreover, this optimization only works on partial programs that can be evaluated on some examples. PROBE's bottom-up search generates complete programs that can always be evaluated on all examples.

3.8 Conclusion and Future work

We have presented a new program synthesis algorithm we dub *guided bottom-up search with just-in-time-learning*. This algorithm combines the pruning power of observational equivalence with guidance from probabilistic models. Moreover, our just-in-time learning is able to bootstrap a probabilistic model during synthesis by leveraging partial solutions, and hence does not require training data, which can be hard to obtain.

We have implemented this algorithm in a tool called PROBE that works with the popular SYGUS input format. We evaluated PROBE on 140 synthesis benchmarks from three different domains. Our evaluation demonstrates that PROBE is more efficient than unguided enumerative search and a state-of-the-art guided synthesizer EUPHONY, and while PROBE is less efficient than CVC4, our solutions are of higher quality.

In future work, we are interested in instantiating PROBE in new application domains. We expect just-in-time learning to work for programs over structured data structures, *e.g.* lists and tree transformations. Just-in-time learning also requires that example specifications cover a range from simple to more complex, so that PROBE can discover short partial solutions and learn from them. Luckily, users seem to naturally provide examples that satisfy this property, as indicated by SYGUS benchmarks whose specifications are taken from StackOverflow. Generalizing these observations is an exciting direction for future work. Another interesting direction is to consider PROBE in the context of program repair, where similarity to the original faulty program can serve

as a prior to initialize the PCFG.

3.9 Acknowledgements

Chapter 3, in full, is a reprint of the material as it appears in Just-in-Time Learning for Bottom-Up Enumerative Synthesis. Barke, Shraddha; Peleg, Hila; and Polikarpova, Nadia. Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA. 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Grounded Copilot: How Programmers Interact with Code-Generating Models

4.1 Introduction

The dream of an "AI assistant" working alongside the programmer has captured our imagination for several decades now, giving rise to a rich body of work from both the programming languages [135, 118, 59, 120] and the machine learning [94, 177, 78] communities. Thanks to recent breakthroughs in large language models (LLMs) [162, 108] this dream finally seems within reach. OpenAI's Codex model [32], which contains 12 billion model parameters and is trained on 54 million software repositories on GitHub, is able to correctly solve 30–70% of novel Python problems, while DeepMind's AlphaCode [108] ranked in the top 54.3% among 5000 human programmers on the competitive programming platform Codeforces. With this impressive performance, large code-generating models are quickly escaping research labs to power industrial programming assistant tools, such as Github Copilot [64].

The growing adoption of these tools gives rise to questions about the nature of AI-assisted programming: *What kinds of tasks do programmers need assistance with? How do programmers prefer to communicate their intent to the tool? How do they validate the generated code to determine its correctness and how do they cope with errors?* It is clear that the design of programming assistants should be informed by the answers to these questions, yet research on these topics is currently scarce. Specifically, we are aware of only one usability study of Copilot,

by [161]; although their work makes several interesting observations about human behavior (which we discuss in detail in Sec. 4.7), ultimately it has a narrow goal of measuring whether Copilot helps programmers in solving stand-alone Python programming tasks. To complement this study and to obtain more generalizable insights that can inform the design of future tools, our work sets out to explore how programmers interact with Copilot in a broader setting.

Our contribution: grounded theory of Copilot-assisted programming. We approach this goal using the toolbox of *grounded theory* (GT) [70], a qualitative research technique that has a long history in social sciences, and has recently been adopted to study phenomena in software engineering [153] and programming languages [112]. GT is designed to build an understanding of a phenomenon from the ground up in a data-driven way. To this end, researchers start from raw data (such as interview transcripts or videos capturing some behavior) and tag this data with categories, which classify and explain the data; in GT parlance, this tagging process is called *qualitative coding*. Coding and data collection must interleave: as the researcher gains a better understanding of the phenomenon, they might design further experiments to collect more data; and as more data is observed, the set of categories used for coding is refined.

In this paper, we present the first grounded theory of how users interact with an AI programming assistant—specifically Github Copilot. To build this theory, we observed 20 participants as they used Copilot to complete several programming tasks we designed. Some of the tasks required contributing to an existing codebase, which we believe more faithfully mimics a realistic software development setting; the tasks also spanned multiple programming languages—Python, Rust, Haskell, and Java—in order to avoid language bias. We then iterated between coding the participants' interactions with Copilot, consolidating our observations into a theory, and adjusting the programming tasks to answer specific questions that came up. The study method is described in detail in Sec. 4.3.

Summary of findings. The main thesis of our theory (Sec. 4.4) is that user interactions with Copilot can be classified into two modes—*acceleration* and *exploration*—akin to the

two systems of thought in dual-process theories of cognition [28, 117], popularized by Daniel Kahneman's "Thinking, Fast and Slow" [93]. In acceleration mode, the programmer already knows what they want to do next, and Copilot helps them get there quicker; interactions in this mode are fast and do not break programmer's flow. In exploration mode, the programmer is not sure how to proceed and uses Copilot to explore their options or get a starting point for the solution; interactions in this mode are slow and deliberate, and include explicit prompting and more extensive validation.

Sec. 4.5 describes two kinds of further analysis of our theory. First, we performed a quantitative analysis of the data collected during the study, comparing prompting and validation behaviors across modes, and quantifying the factors that influence the relative prevalence of each mode. Second, to reinforce our findings, we gathered additional data from five livestream videos we found on YouTube and Twitch, and confirmed that the streamers' behavior was consistent with our theory.

Based on our theory, we provide design recommendations for future programming assistants (Sec. 4.6). For example, if the tool is aware that the programmer is currently in acceleration mode, it could avoid breaking their flow by sticking with only short and high-confidence code suggestions. On the other hand, to aid exploration, the IDE could provide better affordances to compare and contrast alternative code suggestions, or simplify validation of generated code via automated testing or live programming.

4.2 Copilot-Assisted Programming, by Example

Copilot is a programming assistant released by Github in June 2021 [64], and since integrated into several development environments, including Visual Studio Code, JetBrains and Neovim. Copilot is powered by the OpenAI Codex family of models [32], which are derived by fine-tuning GPT-3 [26] on publicly available Github repositories.

In the rest of this section, we present two concrete scenarios of users interacting with

```
# rules are formatted like:
# AB => C
def parse_input(filename):
    with open(filename) as f:
        template, rules = f.read().split("\n\n")
        for rule in rules:
            rule_parts = rule.split(" => ")
```

Figure 4.1: Copilot's end-of-line suggestion appears at the cursor without explicit invocation. The programmer can press `<tab>` to accept it.

Copilot, which are inspired by real interactions we observed in our study. The purpose of these scenarios is twofold: first, to introduce Copilot's UI and capabilities, and second, to illustrate the two main interaction modes we discovered in the study.

4.2.1 Copilot as Intelligent Auto-Completion

Axel, a confident Python programmer, is solving an Advent of Code [171] task, which takes as input a set of rules of the form `AB => C`, and computes the result of applying these rules to a given input string. He begins by mentally breaking down the task into small, well-defined subtasks, the first of which is to parse the rules from the input file into a dictionary. To accomplish the first subtask, he starts writing a function `parse_input` (Fig. 4.1). Although Axel has a good idea of what the code of this function should look like, he thinks Copilot can help him finish it faster and save him some keystrokes and mental effort of recalling API function names. To provide some context for the tool, he adds a comment before the function definition, explaining the format of the rules.

As Axel starts writing the function body, any time he pauses for a second, Copilot's grayed-out *suggestion* appears at the cursor. Fig. 4.1 shows an example of an *end-of-line suggestion*, which only completes the current line of code. In this case, Copilot suggests the correct API function invocation to split the rule into its left- and right-hand sides. To come up with this suggestion, Copilot relies on the *context*, *i.e.* some amount of source file content preceding the cursor, which can include both code and natural language comments, as is the case in our example.

Because the suggestion in Fig. 4.1 is short and closely matches his expectations, Axel

only takes a fraction of a second to examine and accept it, without ever leaving his state of flow. Throughout the implementation of `parse_input`, Axel might see a dozen of suggestions, which he quickly accepts (by pressing `<tab>`) or rejects (by simply typing on). Some of them are larger, *multi-line suggestions*, but Axel still seems to be able to dispatch them quickly by looking for patterns, such as expected control flow and familiar function names. We liken this kind of interaction with Copilot to the fast *System 1* in dual-process theories of cognition [28], which is characterized by quick, automatic, and heuristic decisions.

4.2.2 Copilot as an Exploration Tool

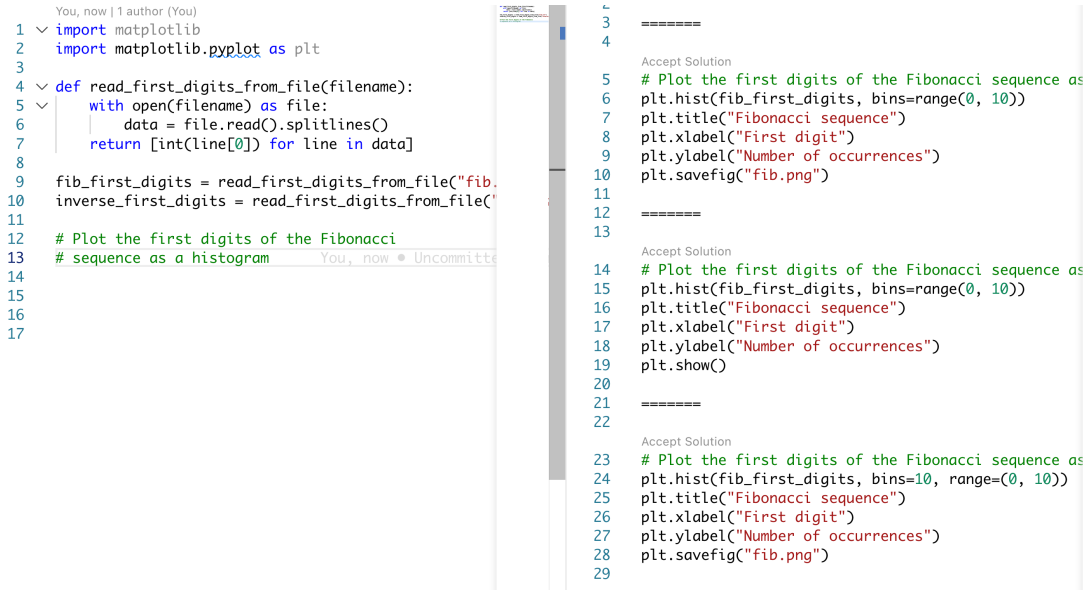


Figure 4.2: The user writes an explicit comment prompt (lines 12–13 on the left) and invokes Copilot's multi-suggestion pane by pressing `<ctrl> + <enter>`. The pane, shown on the right, displays up to 10 unique suggestions, which reflect slightly different ways to make a histogram with `matplotlib`.

Emily is new to data science, and wants to visualize a dataset as a histogram. While she is familiar with Python, she is not familiar with the plotting library `matplotlib`. As a result, she does not know how to approach this task: not only which API functions to call, but also how to decompose the problem and the right set of abstractions to use. Emily decides to use Copilot to explore solutions.

Emily explicitly *prompts* Copilot with a natural-language comment, as shown in lines

12–13 of Fig. 4.2. Moreover, since she wants to explore multiple options, she presses `<ctrl> + <enter>` to bring up the *multi-suggestion pane*, which displays up to 10 unique suggestions in a separate pane (shown on the right of Fig. 4.2). Emily carefully inspects the first three suggestions; since all of them have similar structure and use common API calls, such as `plt.hist`, she feels confident that Copilot understands her task well, and hence the suggestions can be trusted. She copy-pastes the part of the first suggestion she likes best into her code; as a side-effect, she gains some understanding of this part of the `matplotlib` API, including alternative ways to call `plt.hist`. To double-check that the code does what she expects, Emily runs it and inspects the generated histogram. This is an example of *validation*, a term we use broadly, to encompass any behavior meant to increase user's confidence that the generated code matches their intent.

When faced with an unfamiliar task, Emily was prepared to put deliberate effort into writing the prompt, invoking the multi-suggestion pane, exploring multiple suggestions to select a suitable snippet, and finally validating the generated code by running it. We liken this, second kind of interaction with Copilot to the slow *System 2*, which is responsible for conscious thought and careful, deliberate decision-making.

4.3 Method

Participants. We developed our theory through a user study with 20 participants (15 from academia and 5 from industry). We recruited these participants through personal contacts, Twitter, and Reddit. Nine of the participants had used Copilot to varying degrees prior to the study. Participants were not paid, but those without access to Copilot were provided access to the technical preview for continued use after the study concluded. Tab. 4.1 lists relevant information about each participant. We asked each participant to select a statement best describing their level of experience with possible target languages, with options ranging from "I have never used Python", to "I use Python professionally" (from least-to-most, used in Tab. 4.1: Never, Occasional, Regular, and Professional). We screened out participants who had never used the

Table 4.1: Participants overview. PCU: Prior Copilot Usage. We show the language(s) used on their task, their usage experience with their task language (Never, Occasional, Regular, Professional), whether they had used Copilot prior to the study, their occupation, and what task they worked on.

ID	Language(s)	Language Experience	PCU	Occupation	Task
P1	Python	Professional	Yes	Professor	Chat Server
P2	Rust	Professional	No	PhD Student	Chat Client
P3	Rust	Occasional	No	Professor	Chat Client
P4	Python	Occasional	Yes	Postdoc	Chat Server
P5	Python	Regular	No	Software Engineer	Chat Client
P6	Rust	Professional	Yes	PhD Student	Chat Server
P7	Rust	Professional	No	Software Engineer	Chat Server
P8	Rust	Professional	No	PhD Student	Chat Server
P9	Rust[1]	Occasional	No	Undergraduate Student	Benford's law
P10	Python	Occasional	No	Undergraduate Student	Chat Client
P11	Rust+Python	Professional + Professional	Yes	Cybersecurity Developer	Benford's law
P12	Rust+Python	Professional + Occasional	Yes	Software Engineer	Benford's law
P13	Rust+Python	Regular + Occasional	Yes	PhD Student	Benford's law
P14	Python	Professional	No	PhD Student	Advent of Code
P15	Python	Professional	Yes	PhD Student	Advent of Code
P16	Haskell	Professional	No	PhD Student	Advent of Code
P17	Rust	Professional	Yes	Founder	Advent of Code
P18	Java	Occasional	No	PhD Student	Advent of Code
P19	Python	Occasional	No	PhD Student	Advent of Code
P20	Haskell	Occasional	Yes	PhD Student	Advent of Code

target language. We choose a qualitative self-assignment of experience as other common metrics, such as years-of-experience, can be misleading. For example, a professor having used Rust occasionally over eight years is arguably less experienced than as a software engineer using Rust all day for a year.

User protocol. To study participants using Copilot, we gave them a programming task to attempt with Copilot's help. Over the course of an hour a participant was given a small training task to familiarize them with Copilot's various usage models (*i.e.* code completion, natural language prompt, and the multi-suggestion pane). During the core task—about 20-40 minutes—a participant was asked to talk through their interactions with Copilot. They were encouraged to work Copilot into their usual workflow, but they were not required to use Copilot. After the task, the interviewer asked them questions through a semi-structured interview; these questions as well as the tasks are available in our supplementary package. The entire session was recorded and transcribed to use as data in our grounded theory.

Grounded Theory Process. Grounded Theory (GT) takes qualitative data and produces a theory in an iterative process, first pioneered by [70]. As opposed to evaluating fixed, a priori hypotheses, a study using the GT methodology seeks to generate new hypotheses in an overarching theory developed without prior theoretical knowledge on the topic. A researcher produces this theory by constantly interleaving data collection and data analysis. GT has diversified into three primary styles over the past half-century. We follow the framework laid out by Strauss and Corbin [154], commonly called *Straussian Grounded Theory* [153]. We describe our process below.

We began our study with the blank slate question: "How do programmers interact with Copilot?" Our bimodal theory of acceleration and exploration was not yet formed. During each session, we took notes to guide our semi-structured interview. After each session, we tagged any portion of the recording relevant to Copilot with a note. We took into account what the participant said, what they did, and their body language. For example, we initially tagged an instance where P2 was carefully examining and highlighting part of a large Copilot suggestion as "validating sub-expression". Tagging the data in this way is called *(qualitative) coding*; and doing so without a set of predefined codes is called *open coding* in Straussian GT. The first two authors coded the first two videos together, to agree on a coding style, but later data were coded by one and discussed by both.

By the end of the eighth session, we began to see patterns emerging in our data. We noticed two distinct patterns in our codes which eventually crystallized into our acceleration and exploration modes. During this phase of analysis, we aggregated our codes to understand the *conditions* when a participant would enter acceleration or exploration, and the *strategies* a participant deployed in that mode. For example, we realized that if a programer can decompose a problem, then they often ended up in acceleration (details in Sec. 4.4.1). Once in this acceleration mode, programmers would validate a suggestion by a kind of visual "pattern matching" (details in Sec. 4.4.1). This process of aggregating and analyzing our codes form the *axial coding* phase of GT.

After the eighth session, we created a new task to specifically test our emerging theory. This process of testing aspects of a theory-in-progress is known in GT as *theoretical sampling*. After gathering sufficient data on that third task, we created a fourth task to investigate one final aspect of our theory (validation of Copilot's suggestions). In the second half of the study, we linked together our codes and notes into the final bimodal theory we present, in what Straussian GT calls *selective coding*. At the 20th participant, we could fit all existing data into our theory and no new data surprised us. Having reached this point of *theoretical saturation*, we concluded our GT study.

Tasks. The list of all four tasks and their descriptions can be found in Tab. 4.2. Our tasks evolved over the course of the study. We started with the "Chat Server" and "Chat Client" pair of tasks, meant to emulate working on a complex project, with a shared library and specialized APIs. These two initial tasks required contributing to an existing codebase we created, which implements a secure chat application. The first task, Chat Server, asked participants to implement the server backend, focusing on its "business logic". We provided most of the networking code, and the participant's task was to implement the log-in, chat, and chat-command functionality (*e.g.* `/quit` to quit). The complementary task Chat Client focused on the client side of the chat application. Here, we provided no networking code so the participant had to figure out how to use the often unfamiliar socket API. We also required using a custom cryptographic API we implemented, in order to ensure that some part of the API was unfamiliar both to the participant and to Copilot.

To investigate the acceleration and exploration modes further, we created the "Benford's Law"[2] task. This task had two parts, to separately investigate the acceleration and exploration modes we found. In the first half, the participant implements an efficient Fibonacci sequence generator. We believed that all participants would be familiar with the algorithm, and hence would accelerate through this half of the task, allowing us to more deeply characterize the

[2]Benford's Law says that in natural-looking datasets, the leading digit of any datum is likely to be small. It is useful as a signal for finding fraudulent data.

Table 4.2: The four programming tasks used in our study and their descriptions. Task LOC is the lines of code in the provided code and Solution LOC are the number of lines in our canonical solutions.

Task	Language(s)	Description	Task LOC	Solution LOC
Chat Server	Python/Rust	Implement core "business logic" of a chat application, involving a small state machine.	253/369	61/83
Chat Client	Python/Rust	Implement networking code for a chat application, using a custom cryptographic API and standard but often unfamiliar socket API.	262/368	52/84
Benford's Law	Rust & Python	Use Rust to generate two sequences—the Fibonacci sequence and reciprocals of sequential natural numbers; then plot these sequences using Python's `matplotlib`.	9	35
Advent of Code	Python/Rust/ Haskell/Java	Implement a string manipulation task from a programming competition.	2-18	29-41

acceleration mode. In the second half, they plotted this sequence and another ($\frac{1}{2}, \frac{1}{3}, ..., \frac{1}{180}$ as `floats`) using `matplotlib`; this sub-task is used as the example in Sec. 4.2.2. Our participants were not confident users of the plotting library's API, so they needed to turn to some external resource to complete the task. This half stressed the code exploration part of our theory. In addition, our Benford's Law task asked participants to complete the first half in Rust and the second half in Python. This division gave us within-participant information on how different languages impact Copilot usage.

Our fourth task was a string manipulation problem inspired by the 2021 edition of Advent of Code (this task is used as the example in Sec. 4.2.1). We wanted to collect more data about how programmers validate suggestions from Copilot, and this task was a good fit because it comes with a test case and a very precise description, and also has two independent sub-tasks, so it provided several options for checking solutions at different levels of granularity. The data we collected rounded out our hypotheses about validation (Sec. 4.4.1, Sec. 4.4.2).

4.4 Theory

Through our grounded theory analysis, we identified two main modes of developer interactions with Copilot: *acceleration* and *exploration*. In acceleration mode, a programmer

uses Copilot to *execute* their planned code actions, by completing a logical unit of code or a comment. Acceleration works within user's sense of flow. For example, recall how in Sec. 4.2.1 Axel accepted Copilot's suggestion of `rule.split(" => ")`, knowing it was what he wanted to type anyways. This is a characteristic example of acceleration, where Copilot was helping him program faster.

In exploration mode, a programmer relies on Copilot to help them *plan* their code actions. A programmer may use Copilot to assist with unfamiliar syntax, to look up the appropriate API, or to discover the right algorithm. In Sec. 4.2.2, when Emily was searching for the right set of `matplotlib` calls, she was considering alternatives, gaining confidence in the API, and simply trying to learn how to finish her task. All of these intentions are part of the exploration mode when using Copilot. We found that programmers alternate between these two modes as they complete their task, fluidly switching from one mode to the other.

In this section, we systematize our observation of each mode: *acceleration* (Sec. 4.4.1) and *exploration* (Sec. 4.4.2). For each mode, we start with identifying the *conditions* that lead the participant to end up in that mode, and then proceed to describe common *strategies* (*i.e.* behavioral patterns) we observed in that mode. Each numbered subsection (*e.g.* Sec. 4.4.1) is a hypothesis deriving from our top-level bimodal theory. Each named paragraph heading is an aspect of that hypothesis.

4.4.1 Acceleration

Acceleration is characterized by the programmer being "in the zone" and "driving" the development, while occasionally relying on Copilot to complete their thought process. A programmer will often accept a Copilot suggestion without much comment and keep on going without losing focus. In this interaction mode, programmers tend to think of Copilot as an intelligent autocomplete that just needs to complete their line of thought. This idea was well put by P13 who said:

> ”I think of Copilot as an intelligent autocomplete... I already have the line of code in mind and I just want to see if it can do it, type it out faster than I can.”

P15 added to this, calling Copilot “more or less an advanced autocomplete”.

Programmers Use Acceleration after Decomposing the Task. We found that the main causal condition for a participant to end up in acceleration mode is being able to decompose the programming task into *microtasks*. We define a microtask to be a participant-defined task with a well-understood and well-defined job. For example, when P16 was working on the Advent of Code task, they created two separate microtasks to parse the input and to compute the output. Because they understood these microtasks well, they wrote a type signature and used descriptive names for each of them; as a result, Copilot was adept at completing these microtasks for them. Another illustrative example is our Benford’s Law task, which was explicitly designed to have a familiar and an unfamiliar subtask. In the first subtask, participants were asked to implement a fast Fibonacci function. All four participants were familiar with the Fibonacci sequence and knew how to make it efficient. As a result, all of them were able to use Copilot to accelerate through this familiar microtask. P14 explicitly noted:

> “I think Copilot would be more helpful in cases where there are a lot of tedious subtasks which requires less of thinking and more of just coding.”

We observed that language expertise or familiarity with Copilot seem to play less of a role in determining whether a participant would engage in acceleration, compared to their understanding of the *algorithm* for solving the task. For example, P4 was not very comfortable with Python, but they knew what needed to be done in their task algorithmically, and so were able to break it down into microtasks, leading to acceleration. That said, we do observe that language experts and prior Copilot users spend a larger proportion of their total interaction time in acceleration mode; we present quantitative data supporting this observation in Sec. 4.5.

Programmers Focus on Small Logical Units. Participants who interacted with Copilot

in acceleration mode would frequently accept end-of-line suggestions. These were often function calls or argument completions. For example, when P1 wanted to send a message to a client connection object in the Chat Client task, they typed `client_conn.se` and immediately accepted Copilot's suggestion `client_conn.send_message()`. This behavior was seen across all the four tasks when participants were in acceleration mode. For a microtask of parsing file input, P15 wanted to spilt the data based on spaces so they typed `data = x.` to which Copilot correctly suggested `data = x.split("") for x in data`. Participants would happily accept these end-of-line completions with reactions like "Yes that's what I wanted!" and "Thank you Copilot!"

When a programmer is focused on a logical unit of code, they want suggestions *only for that unit*. When they are writing a print statement, they prefer to get a suggestion to the end of the statement. When writing a snippet to message all connected clients, they might instead prefer an entire `for` loop, *but not more*. For example, at one point P8 was focused on a single call to the `startswith` function, but Copilot suggested a large piece of code; P8 reacted with "that's way more than what I needed!" and went on to delete everything but the first line `if msg.startswith('/')`.

The size of a logical unit differs based on the language and context. In an imperative language, this is most often a line of code. However, in a functional language like Haskell, logical units appear to be smaller. P16 said that "in Haskell it just needs to suggest less. [It should] give me the next function I'm going to compose and not the whole composition chain."

Long Suggestions Break Flow. In acceleration mode, long, multi-line suggestions are at best dismissed out of hand and at worst distract the programmer away from their flow.

Upon getting a 16-line suggestion and after just four seconds of review P6 uttered: "Oh God, no. Absolutely not". When P6 got other large suggestions, they would exclaim, "Stop it!", and continue to program as before. This participant also made use of the `<esc>` key binding to actively dismiss a suggestion they did not care for.

On the other hand, many programmers felt "compelled to read the [suggested] code"

(P16) and noted that reading long suggestions would often break their flow. As P1 puts it:

> "When I'm writing, I already have in mind the full line and it's just a matter of transmitting to my fingertips, and to the keyboard. But when I have those mid-line suggestions and those suggestions are not just until the end of line, but actually a few more lines, that breaks my flow of typing. So instead of writing the full line, I have to stop, look at the code, think whether I want this or not."

This sentiment was echoed by multiple participants: P11 was "distracted by everything Copilot was throwing at [them]"; P7 was "lost in the sauce" after analyzing a long suggestion; P17 felt "discombobulated", and others (P8, P11) made similar comments. P16 put it eloquently:

> "I was about to write the code and I knew what I wanted to write. But now I'm sitting here, seeing if somehow Copilot came up with something better than the person who's been writing Haskell for five years, I don't know why am I giving it the time of day."

Such distractions cause some programmers to give up on the tool entirely: P1, P6, and P15 all had Copilot disabled prior to the study—having had access for several months—and they all cited distractions from the always-on suggestions as a factor.

Programmers Validate Suggestions by "Pattern Matching". In order to quickly recognize whether a suggestion is worthwhile, participants looked for the presence of certain keywords or control structures. The keywords included function calls or variable names that they expected should be part of the solution. P1 explicitly stated that the presence or absence of certain keywords would determine whether the suggestion was worth considering.

Most other programmers who commented on how they validated suggestions in acceleration mode mentioned control structures (P4, P17, P19). P4, for instance, immediately rejected an iterative suggestion because they strongly preferred a recursive implementation. On one occasion, Copilot suggested code to P6 when they already had an idea of what shape that code should take; they described their validation process in this instance as follows:

> "I have a picture in mind and that picture ranges from syntactic or textual features—like a literal shape in words—to semantic about the kind of methods that are being invoked, the order in which they should be invoked, and so on. When I see a suggestion, the closer that suggestion is to the mental image I hold in my head, the more likely I am to trust it."

These participants appear to first see and understand control-flow features before understanding data or logic flow features. This is consistent with previous findings dating back to FORTRAN and COBOL programming [129], where programmers briefly shown small code snippets could best answer questions about control flow compared to data- or logic-flow.

Programmers Are Reluctant to Accept or Repair Suggestions. Participants in acceleration mode end up quickly rejecting suggestions that don't have the right patterns. Suggestions that are almost-correct were accepted if a small repair was obvious to the participant. P1 accepted a small inline suggestion which had a call to `handshake()` function, checked if it existed, and since it did not, they made a minor modification, changing the function name to `do_dh_handshake()`. The entire accept-validate-repair sequence seemed to occur without interrupting their state of flow. P1, P4 would often accept similar-looking function names but double check if they actually existed:

> "Each time it uses something else from the context, I usually double check, like in this case it was very similar so I could have been fooled, and each time this happens it reinforces the need to check everything just to see if it has the proper names."

Although programmers tend to dismiss code that does not match their expectations, sometimes Copilot's suggestion makes them aware of a corner case they have not yet considered. P4 saw Copilot write an inequality check while working on the Chat Server task, and they said that they "probably wouldn't have remembered on their first run through to check that [clients] are distinct". Both P6 and P8, working in Rust on the Chat Server, noticed that Copilot used a partial function `.unwrap()`. When asked about this, P8 said:

> "Copilot suggested code to handle it in one case and now I'm going to change it around to handle the other case as well."

4.4.2 Exploration

In the previous section we focused on the use of Copilot when the programmer has a good idea for how to approach the task. But what if they do not? In that case they might use Copilot to help them get started, suggest potentially useful structure and API calls, or explore alternative solutions. All of these behaviors fit under what we call exploration mode. Exploration is characterized by the programmer letting Copilot "drive", as opposed to acceleration, where the programmer is the driver. In the rest of this section, we first describe the conditions that lead programmers to enter exploration mode, and then we characterize the common behaviors in that mode.

Programmers Explore when Faced with Novel Tasks or Unexpected Behavior. Recall that most often the programmer ended up in acceleration mode once they had successfully decomposed the programming task into a sequence of steps (Sec. 4.4.1); dually, when the programmer was uncertain how to break down the task, they would often use Copilot for code exploration. P4 said:

> "Copilot feels useful for doing novel tasks that I don't necessarily know how to do. It is easier to jump in and get started with the task".

Not knowing where to start was one of two primary ways we observed participants begin an exploration phase of their study. The other way participants (P11, P13, P14) began exploration was when they hit some code that does not work as expected, regardless of the code's provenance. They would try a variety of prompting and validation strategies to attempt to fix their bug.

Programmers Explore when They Trust the Model. A participant's level of confidence and excitement about code-generating models was highly correlated with whether and to which extent they would engage in exploration. During the training task, Copilot produced a large,

correct suggestion for P18; they exclaimed, “I’m not gonna be a developer, I’m gonna be a guy who comments!” This level of excitement was shared among many of our participants early in the task, like P7 saying, “it’s so exciting to see it write [code] for you!”. Those participants who were excited about Copilot would often let the tool drive before even attempting to solve the task themselves.

Sometimes, such excessive enthusiasm would get in the way of actually completing a task. For example, P10 made the least progress compared to others on the same task; in our post-study interview, they admitted that they were, “a little too reliant on Copilot”:

> “I was trying to get Copilot to do it for me, maybe I should have given smaller tasks to Copilot and done the rest myself instead of depending entirely on Copilot.”

This overoptimism is characteristic of the misunderstanding users often have with program synthesizers. P9 and P10 were both hitting the *user-synthesizer gap*, which separates what the user expects a program synthesizer to be capable of, and what the synthesizer can actually do [59].

Programmers Explicitly Prompt Copilot with Comments. Nearly every participant (P2, P3, P4, P5, P7, P8, P10, P11, P12, P13, P14, P17, P18, P19, P20) wrote at least one natural language comment as a prompt to Copilot, specifically for an exploratory task.

Programmers prefer comment prompts in exploration. Programmers felt that natural language prompts in the form of comments offered a greater level of control than code prompts (P17). P2 told us that, “writing a couple of lines [of comments] is a lot easier than writing code.” This feeling of being more in control was echoed by P5 who said:

> “I think that the natural language prompt is more cohesive because it’s interruptive to be typing out something and then for Copilot to guess what you’re thinking with that small pseudocode. It’s nice to have a comment that you’ve written about your mental model and then going to the next line and seeing what Copilot thinks of that.”

Programmers write more and different comments when using Copilot. Participants seem to distinguish between comments made for themselves and Copilot. In the words of P6, "The kind of comments I would write to Copilot are not the kind of comments I would use to document my code." P2, P3, P5, P12, and P19 all told us that the majority of their comments were explicitly meant for Copilot. P7 was the sole exception: they wrote comments to jot down their design ideas saying, "I'm writing this not so much to inform Copilot but just to organize my own thoughts"; they added that being able to prompt Copilot using those comments was a nice side effect.

Participants were willing to invest more time interacting with Copilot via comment prompts in exploration mode. They would add detailed information in the comments in the hope that Copilot would have enough context to generate good suggestions (P2, P3). They would rewrite comments with more relevant information if the suggestions did not match their expectations, engaging in a conversation with Copilot. P2 and P6 wished they had a "community guide" (P2) on how to write comments so that Copilot could better understand their intent.

Further, in our interviews, multiple people described their usual commenting workflow as post-hoc: they add comments after completing code. Hence, the participants were willing to change their commenting workflow to get the benefits of Copilot.

Programmers frequently remove comments after completing an interaction with Copilot. Many participants (P3, P4, P7, and P8) would repeatedly delete comments that were meant for Copilot. P19 said that cleaning up comments written for Copilot is essential:

> "I wrote this comment to convert String to array just for Copilot, I would never leave this here because it's just obvious what it's doing. [...] These comments aren't adding value to the code. I think you also have to do like a comment cleanup after using Copilot."

Programmers are Willing to Explore Multiple Suggestions. In exploration mode, we often saw participants spend significant time foraging through Copilot's suggestions in a way

largely unseen during acceleration. This included using the *multi-suggestion pane*, both for its primary intended purpose—selecting a single suggestion out of many—and for more creative purposes, such as cherry-picking snippets from multiple suggestions, API search, and gauging Copilot's confidence in a code pattern.

Participants tend to use the multi-suggestion pane when faced with an exploratory task (P2, P4, P5, P7, P10, P12–20). They would either write a comment prompt or a code prompt before invoking the multi-suggestion pane. This enabled participants to explore alternate ways to complete their task while also providing an explicit way to invoke Copilot. P10, P15, P19 preferred the multi-suggestion pane over getting suggestions inline in all cases. P15 said:

> "I prefer multiple suggestions over inline because sometimes the first solution is not what I want so if I have something to choose from, it makes my life easier."

Some only occasionally got value from the multi-suggestion pane. P6 said that:

> "If I think there's a range of possible ways to do a task and I want Copilot to show me a bunch of them I can see how this could be useful."

Similar to P6, P14 and P17 preferred the multi-suggestion pane only while exploring code as it showed them more options. Yet others turned to the multi-suggestion pane when Copilot's always-on suggestions failed to meet their needs.

Programmers cherry-pick code from multiple suggestions. Participants took part of a solution from the multi-suggestion pane or combined code from different solutions in the pane. P2, P3, P4, P5, P18 often accepted only interesting sub-snippets from the multi-suggestion pane. For example, P18 forgot the syntax for declaring a new `Hashmap` in Java, and while Copilot suggested a bunch of formatting code around the suggestion, P18 only copied the line that performed the declaration. P2 went ahead to combine interesting parts from more than one suggestion stating:

> "I mostly just do a deep dive on the first one it shows me, and if that differs from my expectation, for example when it wasn't directly invoking the handshake function, I specifically look for other suggestions that are like the first one but do that other thing correctly."

Programmers use the multi-suggestion pane in lieu of StackOverflow. When programmers do not know the immediate next steps in their workflow, they often write a comment to Copilot and invoke the multi-suggestion pane. This workflow is similar to how programmers already use online forums like StackOverflow: they are unsure about the implementation details but they can describe their goal. In fact, P12 mentioned that they were mostly using the multi-suggestion pane as a search engine during exploration. P4 often used Copilot for purely syntactic searches, for example, to find the `x in xs` syntax in Python. P15 cemented this further:

> "what would have been a StackOverflow search, Copilot pretty much gave that to me."

Participants emphasized that the multi-suggestion pane helped them use unfamiliar APIs, even if they did not gain a deep understanding of these APIs. P5 explains:

> "It definitely helped me understand how best to use the API. I feel like my actual understanding of [the socket or crypto library] is not better but I was able to use them effectively."

Programmers use the multi-suggestion pane to gauge Copilot's confidence. Participants assigned a higher confidence to Copilot's suggestions if a particular pattern or API call appeared repeatedly in the multi-suggestion pane. Participants seemed to think that repetition implied Copilot was more confident about the suggestion. For example, P5 consulted Copilot's multi-suggestion pane when they were trying to use the unfamiliar socket library in Python. After looking through several suggestions, and seeing that they all called the same method, they accepted the first inline suggestion. When asked how confident they felt about it, P5 said:

> "I'm pretty confident. I haven't used this socket library, but it seems Copilot has seen this pattern enough that, this is what I want."

P4 had a similar experience but with Python syntax: they checked the multi-suggestion pane to reach a sense of consensus with Copilot on how to use the `del` keyword in Python.

Programmers suffer from cognitive overload due to multi-suggestion pane. P1, P4, P6, P7 and P13 did not like the multi-suggestion pane popping up in a separate window stating that it added to their cognitive load. P4 said that they would prefer a modeless (inline) interaction, and P6 stated:

> "Seeing the code in context of where it's going to be was way more valuable than seeing it in a separate pane where I have to draw all these additional connections."

P13 spent a considerable amount of time skimming and trying to differentiate the code suggestions in the multi-suggestion pane, prompting them to make the following feature request:

> "It might be nice if it could highlight what it's doing or which parts are different, just something that gives me clues as to why I should pick one over the other."

Programmers suffer from an anchoring bias when looking through multiple suggestions. The anchoring bias influences behavior based on the first piece of information received. We observed participants believe that suggestions were ranked and that the top suggestion *must* be closest to their intent (P18). This was also evident through P2's behavior who would inspect the first suggestion more deeply then skim through the rest.

Programmers Validate Suggestions Explicitly. Programmers would validate Copilot's suggestions more carefully in exploration mode as compared to acceleration. Their validation strategies included code *examination*, code *execution* (or testing), relying on IDE-integrated *static analysis* (*e.g.* a type checker), and looking up *documentation*. We look at these techniques in detail.

Examination. Unlike acceleration mode, where participants quickly triage code suggestions by "pattern matching", exploration mode is characterized by carefully examining the details of Copilot-generated code. For example, P19 said that they would “always check [the code] line by line”, and P5 mentioned that their role seemed to have shifted from being a programmer to being a code reviewer: “It’s nice to have code to review instead of write”. Participants found it important to cross-check Copilot’s suggestions just as they would do for code from an external resource. When asked how much they trusted Copilot’s suggestions, P14 said:

> “I consider it as a result I would obtain from a web search. It’s not official documentation, it’s something that needs my examination...if it works it works”

Execution. Code execution was common—occurring in every task by at least one participant—although not as common as examination. In case of the server and client task, participants P3 and P7 would frequently run their code by connecting the client to server and checking if it has the expected behavior. For the Benford’s law task, P11 wrote test cases in Rust using `assert_eq` to check whether the Fibonacci function suggested by Copilot was correct. All participants in the Advent of Code task ran their code to check whether they parsed the input file correctly.

In addition to executing the entire program, some participants used a Read-Eval-Print-Loop (REPL) as a scratchpad to validate code suggestions (P14, P16, P19). P16 used the Haskell REPL throughout the study to validate the results of subtasks. Copilot suggested an `adjacents` function that takes a string and pairs adjacent characters together. P16 validated the correctness of this function by running it on toy input `adjacents "helloworld"`.

Static analysis. In typed languages like Rust, the type checker or another static analyzer frequently replaced validation by execution. For example, P17 did not run their code even once for the Advent of Code task, despite the task being designed to encourage testing. They reasoned that the `rust-analyzer`[3] tool—which compiles and reports type errors in the background—took

[3]https://github.com/rust-lang/rust-analyzer

away the need to explicitly compile and execute the code.

> "In Rust I don't [explicitly] compile often just because I feel like the there's a lot of the type system and being able to reason about state better because mutability is demarcated a lot. But if this were in Python, I would be checking a lot by running in a REPL."

P7 thought it was "cool how you can see the suggestion and then rely on the type checker to find the problems." In general, most participants using statically typed languages relied on IDE support to help them validate code. P6, P8 and P17 relied on Rust analyzer and P6 had this to say:

> "I rely on the Rust compiler to check that I'm not doing anything incorrect. The nice part about being a statically typed language is you can catch all that at compile time so I just rely on Rust analyzer to do most of the heavy lifting for me."

Documentation. Lastly, consulting documentation was another common strategy to explicitly validate code from Copilot. As an example, P11 was trying to plot a histogram in `matplotlib`, but was unsure of the correct arguments for the `plt.hist` function. They accepted a couple of Copilot's suggestions but explicitly went to validate the suggested arguments by reading the documentation within the IDE. Participant P17, who never executed their Rust code, would instead hover over the variables and function names to access API documentation within the IDE. Participants that did not have documentation built into their IDE would turn to a web resource. For example, P14 accepted Copilot's suggestion for parsing file input in the Advent of Code task, and then validated the functionality of `splitlines` by crosschecking with the official Python documentation. P11 also used Google for crosschecking whether the Fibonacci sequence suggested by Copilot was accurate.

Programmers Are Willing to Accept and Edit. Unlike acceleration mode, where participants were quick to dismiss a suggestion that didn't match their expectations, during exploration,

they seemed to prefer deleting or editing code rather than writing code from scratch. When a participant saw a segment of code that they felt they were likely to need in the future, they would hang on to it (P2, P3, P4, P6, P8). P2 was exploring code for one stage of writing a chat server when they saw code needed for a later stage and said: "I'm keeping [that] for later". During their exploration, they accepted a 40 line block of code to add:

> "Eh, I'm just going to accept this. It's close enough to what I want that I can modify it."

P3 said: "I wanna see what [Copilot] gives me, then I'll edit them away". Some participants were able to complete most of their task by accepting a large block of code and then slowly breaking it down. P7 accepted a large block of code early on and iteratively repaired it into the code they needed. P5 had a similar experience and said, "It's nice to have code to review instead of write".

Commonly, participants sought a suggestion from Copilot only to keep the control structure. As a representative example, P8 was writing a message-handling function in Rust, when Copilot produced a 15-line suggestion, containing a `match` statement and the logic of its branches. After examination, P8 accepted the suggestion but quickly deleted the content of the branches, retaining only the structure of the `match`. We saw this many times with P1, P2, P11, P17, P18 as well. P17 said:

> "If I'm in a mode where I want to rip apart a solution and use it as a template then I can look at the multi-suggestion pane and select whichever suits my needs."

Copilot-generated code is harder to debug. On the flip side, participants found it more difficult to spot an error in code generated by Copilot. For example, P13 had to rely on Copilot to interface with `matplotlib`; when they noticed undesired behavior in that code, they said:

> "I don't see the error immediately and unfortunately because this is generated, I don't understand it as well as I feel like I would've if I had written it. I find reading code that I didn't write to be a lot more difficult than reading code that I did write, so if there's any

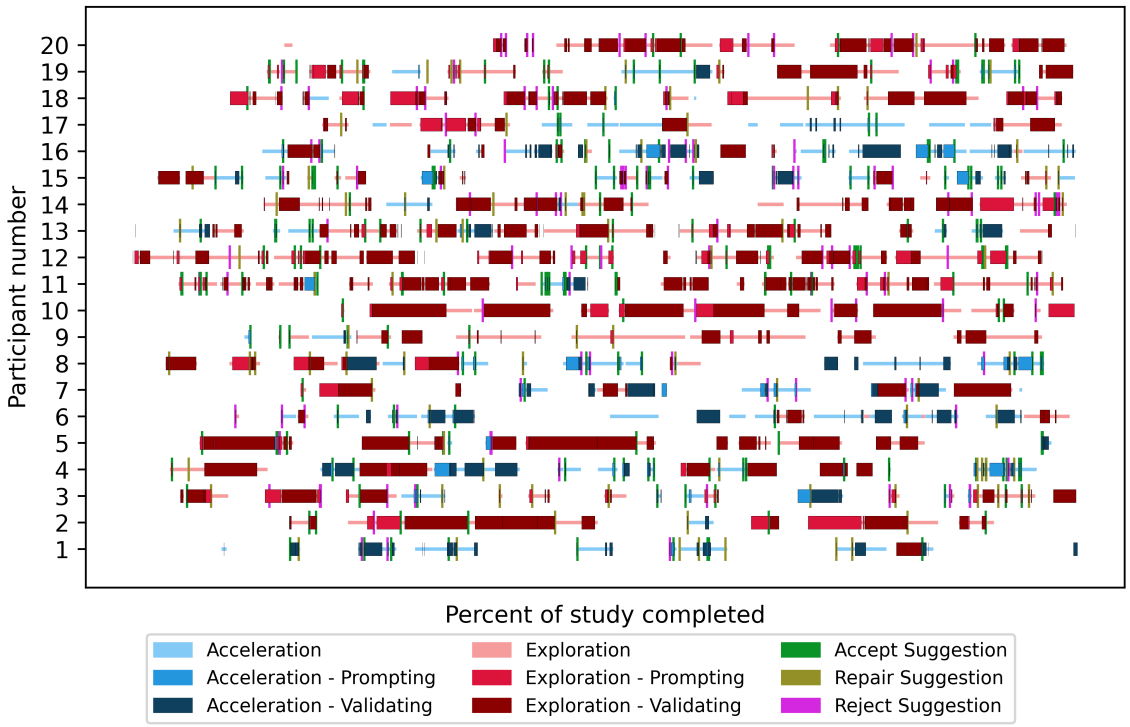


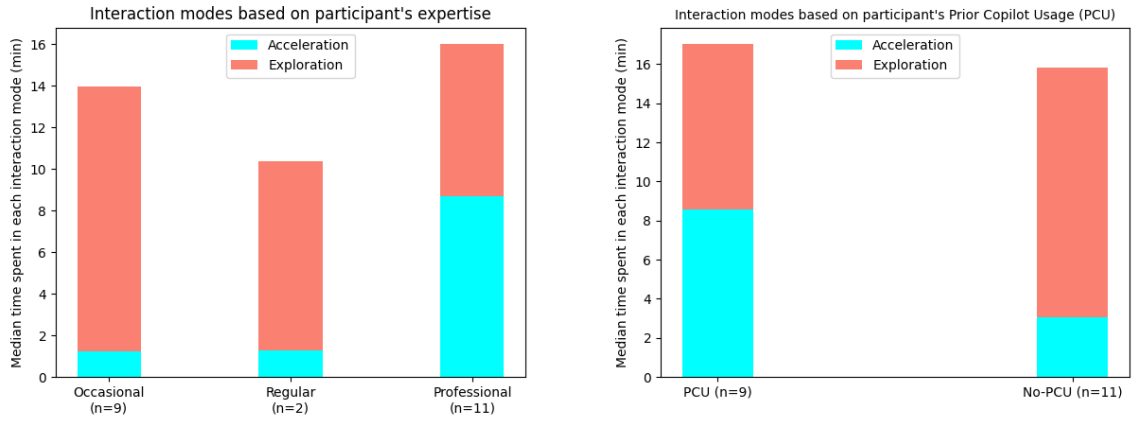
Figure 4.3: Timeline of observed activities in each interaction mode for the 20 study participants. The qualitative codes include different prompting strategies, validation strategies and outcomes of Copilot's suggestions (accept, reject or repair)

> chance that Copilot is going to get it wrong, I'd rather just get it wrong myself because at least that way I understand what's going on much better."

We observed a similar effect with P9, who could not complete their task due to subtly incorrect code suggested by Copilot. Copilot's suggestion opened a file in read-only mode, causing the program to fail when attempting to write. P9 was not able to understand and localize the error, instead spending a long time trying to add more code to perform an unrelated file flush operation.

4.5 Additional Analysis

In this section, we first provide quantitative evidence to support the findings from our grounded theory analysis. We then present the results of a qualitative analysis on five livestream videos to provide additional evidence that further supports our theory.



(a) Grouped by language expertise.

(b) Grouped by prior Copilot usage.

Figure 4.4: Median time spent in acceleration vs exploration mode for different participant groups.

4.5.1 Quantitative Analysis

At the end of our grounded theory analysis, we closed our codebook and re-coded all videos with a fixed set of codes that emerged to be most noteworthy. Fig. 4.3 represents this codeline of the different activities we observed in each of the two interaction modes. The activities include prompting strategies, validation strategies, and the outcomes of Copilot's suggestions *i.e.* whether the participant accepts, rejects, or edits the suggestion. We then performed a quantitative analysis on this codeline to investigate the following questions:

(**Q1**) What factors influence the time spent in each of the two interaction modes?

(**Q2**) What are the prompting strategies used to invoke Copilot in the two interaction modes?

(**Q3**) How do the validation strategies differ across the two interaction modes and by task?

Time Spent in Interaction Modes. The total amount of study time spent by all participants interacting with Copilot in exploration mode (248.6 minutes) is more than twice that in acceleration mode (104.7 minutes). This is not surprising, since exploration is the "slow *System 2*" mode, where each interaction takes longer. At the same time, the ratio of time spent in the two modes is not constant across participants. Below, we investigate which factors influence this ratio,

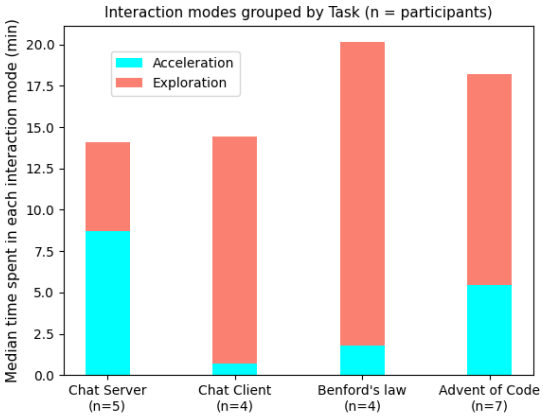


Figure 4.5: Median time spent in acceleration vs exploration mode, grouped by task.

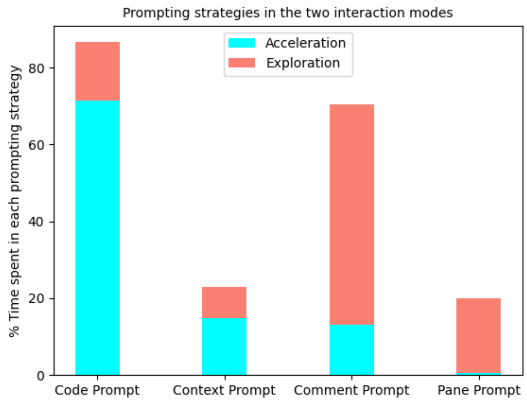


Figure 4.6: Prevalence of prompting strategy as percentage of total prompting time.

including language expertise, prior Copilot usage, the nature of the task, and the programming language.

Language expertise. Fig. 4.4a shows the median time spent in two modes split by the participant's language expertise. We can clearly see that professional participants with the most language expertise spend more time accelerating than the other two groups. This is not surprising, since they are more likely to already know how to solve the task in the given language.

Prior Copilot usage. We can see in Fig. 4.4b that the total interaction time is roughly the same for participants with and without prior Copilot usage. Given roughly the same overall time, prior users spend less time exploring (and more time accelerating) than novice users. We attribute this difference to the effect we observed in Sec. 4.4.2, where novice users have higher expectations of Copilot's ability to solve high-level exploratory tasks.

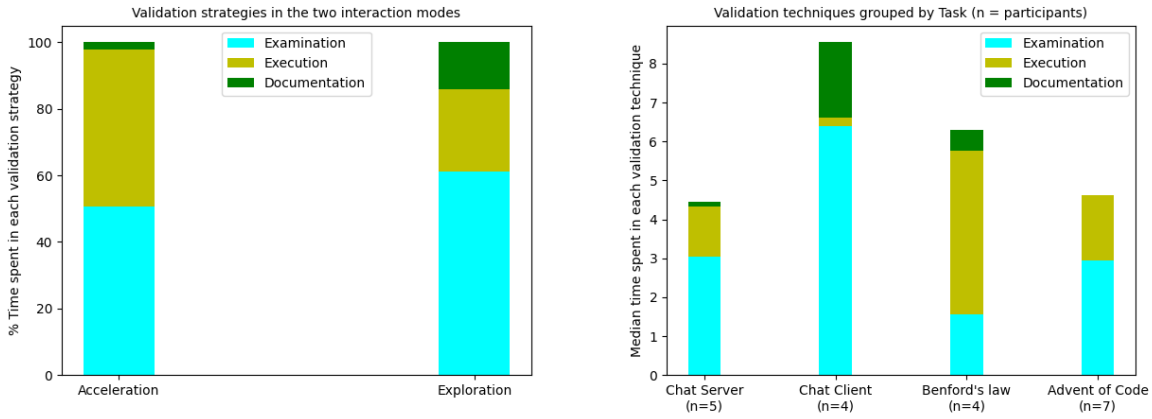
Nature of Task. Fig. 4.5 shows the median time spent in each mode grouped by task. Both Chat Client and Benford's Law prominently feature interaction with unfamiliar APIs; as a result, all participants in these two tasks spent considerably more time in exploration, irrespective of other varying factors such as language expertise and prior Copilot usage. Advent of Code was more algorithmically challenging than the other tasks, and also involved the File I/O API, which was somewhat unfamiliar to participants. Both of these factors pushed participants to explore

but there was more variance in the data than in Chat Client and Benford's Law: for example, P16, who figured out the algorithm early on, spent more time accelerating (15.8 minutes) than exploring (3.4 minutes). Chat Server, on the other hand, involved simple business logic, so participants leaned towards acceleration in this task.

Programming Language. We did not identify any noticeable differences in either total interaction time or ratio of acceleration to exploration between Python and Rust. For the other two languages (Haskell and Java), we have too few data points to make any conclusions.

Prompting Strategies across Interaction Modes. Our codebook identifies four strategies participants use to invoke Copilot: *code prompts*, *context prompts*, *comment prompts*, and the *multi-suggestions pane*. We can cluster the four prompting strategies into two categories: unintentional prompting (Sec. 4.2.1) and intentional prompting (Sec. 4.2.2). Unintentional prompting involves participants invoking Copilot without explicitly meaning to. For example, with *code prompts*, the participant is often simply writing code when Copilot pops up a suggestion to complete their partially written line of code. *Context prompts* are those where Copilot generates suggestions even when the participant is not actively writing code. From the language model perspective, these two kinds of prompts are indistinguishable but we consider them distinct from the user interaction perspective. Intentional prompting involves explicit intent from the participant. This can be in the form of writing a natural-language comment intended for Copilot (Sec. 4.4.2) or invoking the multi-suggestions pane by pressing `<ctrl> + <enter>` (Sec. 4.4.2).

Fig. 4.6 shows the aggregate percentage of times the 20 participants invoked Copilot using the four different prompting strategies. We notice that in acceleration mode, the most commonly used prompting strategy is code prompts (71.4%), with the other unintentional strategy, context prompts, coming in second (15.2%). The multi-suggestions pane is rarely used, which is consistent with our theory, since it would break the participant's flow. In exploration mode, participants intentionally prompt with comments a lot more than in acceleration mode (57.2% vs 13.1%). The percentage of multi-suggestion pane prompts also shoots up in exploration mode as



(a) Aggregate time, split by interaction mode.

(b) Median time, grouped by task.

Figure 4.7: Time spent in different validation strategies.

it provides a rich body of suggestions for participants to explore from.

Validation Strategies across Interaction Modes and Tasks. Recall that Sec. 4.4.2 identified four different validations strategies: *examination*, *execution*, *static analysis*, and consulting the *documentation*. We measured the time participants spent in each of these strategies, with the exception of static analysis, which runs automatically in the background, so it was hard for us to determine precisely when a participant was "using" its results.

Fig. 4.7a shows the percentage of validation time spent in each strategy, split by interaction mode. Predictably, participants spent more time reading *documentation* in exploration mode than in acceleration mode, likely because exploration was commonly used when interfacing with unfamiliar APIs. A somewhat surprising result is that *execution* seems to be more prevalent during acceleration. One reason for this is that during exploration the code is often incomplete and cannot be executed. Another reason is simply that the remaining strategy, *examination*, takes up more time in absolute terms during exploration, as participants carefully examine the code line by line as opposed to making quick decisions via "pattern matching". We conclude that in exploration mode, programmers use validation strategies that *aid comprehension* (careful examination, reading documentation), while in acceleration more, they focus on strategies that provide *rapid feedback* on code correctness (execution).

The nature of the task also has impact on the validation strategies, as shown in Fig. 4.7b. The prevalence of complex and unfamiliar APIs in Chat Client both increases the overall validation time for this task and favors exploratory validation, such as examination and documentation. Interestingly, the task with most time spent in execution is Benford's Law, and not Advent of Code, which was explicitly designed to be easy to test (it came with a test case). We conjecture that Benford's Law was executed so often because it has visual output, which is easy and exciting for programmers to inspect.

4.5.2 Qualitative Analysis of LiveStreams

We gathered additional evidence in the form of five livestream videos to support our theory. We present our findings from a qualitative analysis of these videos in this section.

Data Collection. The livestream videos were taken from Youtube (S1, S2, S4) and Twitch (S3, S5), and involved a developer using Copilot while constantly talking aloud to an audience. S1 and S2 had Copilot turned on to solve Advent of Code tasks in Haskell and C# respectively. S3, S4 and S5 all did web-based programming tasks using Copilot in Javascript, Typescript, HTML, SCSS, and other web languages. For example, S4's task was to build a Go game in Angular. While S1, S2 and S4 had well-defined tasks, S3 and S5 used Copilot for exploratory tasks, in fact, S3 even asked their viewers to suggest random programming tasks for Copilot.

Qualitative Data Analysis. One of the authors coded all the livestream videos with the same closed codebook used to re-code our participant videos in Sec. 4.5.1. We present the results from our qualitative analysis and draw parallels to our bimodal theory of acceleration and exploration.

Acceleration Mode. We observed that when the task was relatively well-defined (S1, S4, S5), acceleration mode was prevalent, consistent with our theory. All streamers used Copilot for end-of-line completions in acceleration mode at least once, accompanied with comments like, "Yeah Copilot knows what I'm trying to do!" In fact, S4 used Copilot only for end-of-line completions and said, "I need to let the AI help more, I'm doing too much stuff myself."

Streamers would only focus on small logical units, for instance, S2 accepted a long suggestion only to retain the structure of a for loop and the condition within. S2 repeated this behavior when they just wanted to fill in the parameters of a function so they ended up deleting everything in a suggestion except the parameters. S1 often used end-of-line completions to complete type signatures in Haskell, which would correspond to a logical unit. As observed in our theory, the streamers would reject long suggestions that broke their flow (S1, S2, S4). S4 exclaimed, "Thank you, that's not what I want" when Copilot suggested an extremely long snippet while they were accelerating. In addition, both S1 and S4 made only minor edits to suggestions accepted in acceleration mode, whereas S1 made relatively major edits to suggestions in exploration.

Exploration Mode. S3 and S5, who worked on exploratory tasks, spent considerably more time in exploration mode than in acceleration. Streamers were willing to write a lot of comments while in exploration mode (S2, S3, S5). S5 tried to use Copilot to generate documentation and said, "as a person who usually writes comments after writing code, Copilot might change the way I code". S3 had an interesting way of prompting Copilot: by writing unusually descriptive function names instead of comments. S3 and S5 often used the multi-suggestion pane as a fallback option when the inline suggestions did not meet their expectations. S3 expected the multiple suggestions to be diverse and was sometimes disappointed when they were not. In addition to using the pane, S5 also explored multiple suggestions inline by pressing tab. We did not observe this behavior in our main study, because neither we nor our participants were aware of this feature.

Validation Strategies. We observed the same validation behavior as seen in our theory in all the livestream videos. After accepting a suggestion, S3 said, "Let's just check if this part works" and S1 echoed, "I think Copilot wrote that for me, let me just check". S1 and S2 constantly validated their code using the test inputs provided by the Advent of Code tasks and also used specific test inputs for debugging code. All streamers spent considerable time in code examination as a validation strategy both inline (S1, S2, S4, S5) and in the multi-suggestion

pane (S3, S5). S2 and S3 referred to API documentation using web search to validate Copilot's code while S4 resorted to reading in-IDE API documentation as a form of validation. S3 and S4 whose tasks involved building a website ran their webpage remotely as a validation strategy.

The blame game of who wrote the buggy code was also observed in the livestreams. While debugging their code, S5 expressed this by saying, "not sure if they are my bugs or Copilot's bugs." S3 had a bug that they were baffled by, turns out it was some residual code from Copilot's suggestion which they forgot to delete. S5 summed up Copilot's behavior as being a "mixed bag, when it understands what I want it feels like it's reading my mind. Otherwise it produces random code." Streamers were generally confident using Copilot for writing boilerplate, repetitive code (S3, S4).

4.6 Recommendations

This section outlines recommendations for how programming assistants could be improved in the future, We classify these suggestions into two categories: improving the way programmers could provide *input* to a future tool, and improving the kinds of *output* the tool could generate.

4.6.1 Better Input

Control over the context. There was general confusion among participants about how Copilot uses their code to provide suggestions. Some participants were unsure how much code Copilot can take into context, for example, P8 theorized a hard limit to the input length: "I think the README is too long and complicated for it to actually extract [helpful information]". Other participants (P8, P10, P15, P18) mentioned they were unsure about which pieces of information Copilot had extracted about their local codebase. Specifically, there appeared to be a broad misconception that commenting out code made it invisible to Copilot, despite those same participants using comment prompts. P20 "assumed it wouldn't be aware of code if [they] commented it out". We also observed participants (P2, P3, P4 P6) comment out code generated

by Copilot in an attempt to get it to generate an alternative suggestion.

Participants that *were* aware of Copilot's sensitivity to context wanted to have more control over that context. Some participants wanted to give Copilot *specific context*: in describing their work outside of the study, P15 mentioned poor suggestions from Copilot and wished they could emphasize a subset of their code (*i.e.* niche libraries they imported), so they could feel more confident that the suggestions were relevant to their code. Others, P4 and P12, wished to query Copilot with a natural-language prompt *without* any code context, just as they would query StackOverflow.

In order to achieve this control, participants wanted Copilot to provide dedicated *syntax*. For example, P2 wanted Copilot to use a specific function, and tried to achieve this by "using the function name in backquotes". P18 asked: "Is there a way to prompt Copilot into suggesting a data structure?" Finally, P4, when looking for examples of using the `del` operation in Python, wanted to explicitly ask Copilot to show only "syntax examples".

Based on these observations, future tools could give programmers ways to customize the context. For example, a future tool could provide a scratchpad to isolate general, StackOverflow-style prompts from the rest of the codebase. It could also provide expert prompt syntax, similar to advanced operators in Google search; for example, including `:use plt.show()` in a comment prompt might restrict the assistant's suggestions to only those snippets using the expression `plt.show()`, like the work of [128]. Finally, programmers would likely appreciate a separate type of comments that make code invisible to the tool.

Cross-language translation. P13 said that they were more familiar with Julia than the task language (Python), and at some point they wrote some Julia code which Copilot then translated to Python. This type of interaction opens up the possibility of users giving prompts in programming languages they are more familiar with. The task for Copilot then becomes a cross-language translation task. It would be interesting to fine-tune Copilot for this particular task, by training it on equivalence classes of syntactic constructs in different programming

languages.

4.6.2 Better Output

Awareness of the interaction mode. Perhaps the most important outcome of our study is the bimodal nature of programmers' interactions with Copilot: they are either in an acceleration or exploration mode. We conjecture that the user experience could be improved if the tool were aware of the current interaction mode and adjusted its behavior accordingly. In acceleration mode, it should not break the programmer's flow (P6 mentioned that they intentionally turned Copilot off because it disrupted their workflow). To this end, the tool should avoid low-confidence suggestions—which are unlikely to be accepted—and long suggestions—which distract the programmer.

Going beyond simply avoiding multi-line suggestions, the tool could be made more aware of how the code is divided into logical units. As we mentioned in Sec. 4.4.1, programmers in acceleration mode focus on a single logical unit of code at a time, which is often one line, but can also be shorter (the next function call in Haskell) or longer (an entire loop). It would be interesting to explore if we can make the scope of Copilot's suggestions match the scope the programmer's current focus. Participants also mentioned that it would be helpful if Copilot gave suggestions more selectively as opposed to being always on. This could be achieved, *e.g.*, by reinforcement learning to obtain a policy for when Copilot should intervene, based on the local context and programmer's actions.

Exploring multiple suggestions. As we mentioned in Sec. 4.4.2, in exploratory searches, programmers commonly used the multi-suggestion pane, but also often got overwhelmed by the results they saw there. Several participants had trouble identifying meaningful differences between the suggestions (P1, P4, P6, P7, P13). This observation motivates the need for a tool that would help programmers explore a large space of suggestions, perhaps similarly to how Overcode [71] supports exploring a space of student solutions to a programming assignment.

Suggestions with holes. Recall from Sec. 4.4.2, that when programmers modify sugges-

tions, they often keep control-flow features and little else, as seen for P1, P2, P8, P11, P17, and P18. Based on this observation, programmers would likely benefit from *suggestions with holes*, where the tool only generates control structures, which users are likely to understand quickly, leaving their bodies for the programmer to fill out (either by hand, or by giving more targeted prompts to the tool). For example, P2 explicitly mentioned that "if [Copilot] gives me a mostly filled out skeleton, I can be the one who fills out holes". Recent work by [78] generated holes in their suggestions where the underlying model had low confidence.

Low-confidence suggestions are not the only motivation for a hole: participants reported feeling frustrated and distracted by large code snippets. When offered these large snippets, some participants felt Copilot was forcing them to jump in to write code before coming up with a high-level architectural design. P4 said:

> "I wrote code as one might read code, rather than the way I might write it which is generally top-down, where I will fill in the control structure and then I'll do the little bits and pieces after I build in the full control structure. It made me jump in to write code instead of the normal way."

P16 normally writes a high-level design first and then gets to function implementations—as the grounded theory from [112] describes of functional programmers. Other participants (P2, P3, P4, P5, P7, P8) also felt Copilot forced significant change on their code authorship process. Based on our observations, future tools should mind how large code blocks can break the user's natural development flow, instead offering code holes for users to fill in when ready.

Always-on validation. Several participants (P2, P14, P16) wished to have better tool support for validating suggestions. For example, P16 wanted to set up property-based testing [39] to run automatically on Copilot suggestions. P14 wished they had *projection boxes* [105], a live programming environment that constantly displays runtime values of relevant variables (usually on a single test input). In the future, IDEs could couple code-generating models with some kind

of always-on validation, in order to make the process of evaluating code suggestions less taxing for the developer.

4.7 Related Work

Usability of Copilot. The closest to our work is the study by [161], which also evaluates Copilot. In our study, we explicitly stepped away from the common comparative setting, where participants are given well-defined stand-alone tasks, and the goal is to collect quantitative data on how well and quickly they complete the tasks, with and without the tool under evaluation. Instead, we chose more open-ended tasks in the context of an existing codebase, which we believe is closer to the real-world use case. Further, instead of skewing quantitative answers to predefined research questions, we chose the grounded theory approach, with the general goal of finding patters in programmers' behavior when they interact with Copilot; we believe this approach is complementary to the quantitative studies. Finally, our usage of multiple languages enables inter-language comparisons and more generalizable conclusions.

On the other hand, [161] also report several qualitative findings. Most of them agree with ours, such as: that Copilot often provides a good starting point for programmers who do not know how to approach the task, that programmers are generally willing to repair code suggestions, but Copilot-generated code is harder to debug. There are also some differences; for example, half of their participants (12/24) said they had trouble understanding and modifying Copilot-generated code, whereas our participants did not seem to share this difficulty; this might be because our study is with more experienced developers: only one participant in our study was an undergraduate student, whereas 10/24 in their study were undergraduates.

Usability of other LLM tools. Beyond Copilot, [91] conducted a user study to analyse the interaction of developers with a natural language to code tool called GenLine. GenLine is similar to Copilot but involves explicitly invoking a command within a text editor. Similar to our findings, developers in their study were willing to rewrite the natural language prompt to

clarify their intent and expressed the need for a syntax to communicate with the model more clearly. However, their findings were mainly centered around prompting strategies whereas we did a more comprehensive analysis of developer interactions with Copilot. Moreover, the tool was not integrated in the participant's daily workflow like in our study. In a similar vein, [178] investigated the usefulness of an NL-to-code plugin previously developed by the same authors [177]. They found no statistically significant difference in task completion times or correctness scores when using the plugin, and the participants' feedback about the plugin was neutral to slightly positive. We conjecture, however, that these findings are not as relevant anymore, thanks to recent breakthroughs in large language models, which significantly increased the quality of generated code. In another related study, [173] interviewed IBM software engineers about their experience with a neural machine translation tool for translating code between programming languages. This study focuses on the engineers' code validation strategies and future UI features that might help with this task, such as confidence highlighting and alternative translations; in this sense, their study is complementary to our work, conducted in the context of a different task (language-to-language translation).

[140] compiled observations from the above user studies and additionally gathered experience reports of programming assistants usage from Hacker News. The compiled observations were similar to what we found—prompting is hard, validation is important, and programmers use assistants for boilerplate, reusable code. There are a few other industrial-grade programming assistants powered by statistical models, such as TabNine [155] and Kite [95], but we are not aware of any research on their usability.

Usability of program synthesis tools. Another approach to code generation, is the more traditional, search-based program synthesis. As program synthesis technology matures, it becomes increasingly common to evaluate the usability of synthesizers on human subjects. Many of these usability studies are for domain-specific synthesizers targeting API navigation [88], regular expressions [187, 186], web scraping [31], or data querying, wrangling, and visual-

ization [46, 166, 189]. These studies usually focus on measuring the tool's effect on task completion rates and times, which is less relevant to our questions. The work on RESL [126] and Snippy [59, 58] include user studies of general-purpose programming-by-example synthesizers for JavaScript and Python. Although both also focus mainly on task completion times, they do make some interesting qualitative observations. For example, [59] observe that one of the main barriers to the usefulness of the synthesizer is the so-called *user-synthesizer gap*, *i.e.* the programmer's overestimation of the synthesizer's capabilities; we observed a similar phenomenon in our study (see Sec. 4.4.2), although it appears to be less prominent in LLM-based tools, since their performance degrades more gradually with the complexity of the task.

[89] study how undergraduate students learned to use six different synthesizers—Copilot among them—with different interaction modes. Not all of the themes they identify are applicable to Copilot, but those that are, are corroborated and explored in more depth in our study. For example, they identify that novice participants would often accept then modify code. We support and extend this (Sec. 4.4.2), adding that this is characteristic of exploration mode, which indeed occurs more commonly in novices.

Grounded Theory for software development. Grounded Theory (GT) has a relatively long history in software-related fields, with its application to software engineering dating as far back as 2004 [29]. [153] provide a survey and a critical evaluation of 93 GT studies in software engineering. Recently, GT has also drawn interest in the programming languages community: [112] study how statically-typed functional programmers write code, and deliver a set of guidelines meant for functional language tool-builders.

4.8 Limitations and Threats to Validity

Our participants worked on tasks of our design, as opposed to their own projects. If they were working in a more familiar codebase and without the time pressure of a study, their interactions could have been different. Moreover, our tasks focused only on code authorship, as

opposed to refactoring, testing, debugging, or other common aspects of software engineering. We consider these beyond the scope of this study, although our participants did occasionally get a chance to test or debug their code.

We recruited 20 participants, with a skew towards those in academia, hardly a representative sample of all programmers. Similarly, although we tried to diversify the type of tasks our participants were solving and the programming languages they were using, other kinds of tasks and languages could have lead to different interactions.

11 of our participants had not used Copilot before the study, and hence might not be representative of regular users of the tool. We gave all participants a 5-minute training task so they could familiarize themselves with Copilot, and yet we observed that first-time users were sometimes over-reliant on Copilot, in a way that prior users were not. We chose to include new users in our study since the majority of programmers in the wild have never used a code-generating model. Meanwhile, our participants who already had access to the tool may have formed a usage pattern (or dis-uage pattern in the case of P6) based on poor experience early in the technical preview, where its behavior may have been rapidly changing. Ideally, we would have liked to observe programmer over a longer period of time, in order to study how their usage patterns changed over time, but this was not feasible given the time constraints of the study.

Finally, the research on code-generating models is progressing very rapidly, and it is possible that new technological breakthroughs will soon render our findings obsolete. That would be a nice problem to have indeed!

4.9 Acknowledgements

Chapter 4, in full, is a reprint of the material as it appears in Grounded Copilot: How Programmers Interact with Code-Generating Models. Barke, Shraddha; James, Michael B.; and Polikarpova, Nadia. Proceedings of the ACM on Programming Languages, Volume 7, Issue OOPSLA1. The dissertation author was a primary investigator and author of this paper.

Chapter 5

Solving Data-centric Tasks using Large Language Models

5.1 Introduction

Code-generating large language models (LLMs) promise to empower end users interested in *data-centric tasks*, ranging from string manipulations in spreadsheets to data cleaning and analysis in computational notebooks. For example, consider the following task on tabular data: given a column with *full names*, generate a new column with *user names*, by combining the first initial and last name, in lowercase. This task can be solved by a Pandas program that: 1) splits the full name into a list of strings, 2) extracts the first and last string from the list, 3) converts both to lowercase and joins the first letter of one string to the other as shown in Fig. 5.1. The challenge in generating this program is that input data rows have varied formats, *e.g.* most rows only have two names ("John Smith"), but some have multiple middle names ("Jake L Woodhall", "Jo Anna Emily Gray"). If an LLM prompt does not include any data or only includes rows with two names, the LLM is more likely to generate a program that does not generalize (*e.g.* one that extracts the last name as the *second* element of the list instead of *last*).

In this paper, we focus on solving such tasks that involve multi-step computations on the input columns to generate additional columns. Towards this goal, we mine StackOverflow to construct a new dataset, dubbed SOFSET, of data-centric tasks, equipped with a natural-language query and a small input table. Using this dataset, we conduct experiments on generating Pandas

programs using GPT-4 and an open-source alternative CODELLAMA, with the goal of analysizing LLMs' sensitivity to the amount of input data provided in the prompt.

Unlike input tables in StackOverflow posts, real-world data tables are often large, hence sending the entire table to the LLM is likely impractical, expensive, or detrimental to performance. *How do we best convey the structure of a large input table to the LLM?* To address this question, we propose a *cluster-then-select* prompting technique that clusters input rows based on their syntactic structure and then selects representative rows from each cluster; *e.g.* in our Fig. 5.1 example, the technique would include a row for each format of middle names. To evaluate this technique, we perform experiments on SOFSET augmented with larger input tables extracted from Kaggle.

In summary, this paper contributes:

- a real-world dataset of complex tasks for evaluating data-centric code generation;
- a *cluster-then-select* technique for selecting rows to prompt with, from large input tables;
- an analysis that shows LLMs are sensitive to the data quantity, choice and position of rows.

5.2 Related Work

Large language models for tabular data Code-generating LLMs like CODEX [33], INCODER [63] and PALM [38] have been fine-tuned for code-specific tasks and adapted for data-centric domains like SQL [158, 133]. [109] explore the ability of models like BERT to perform entity matching on tabular data. [119] use GPT3.5 for data cleaning, error detection and entity matching tasks. [83] focus on tabular classification tasks and explore parameter-efficient LLM tuning.

Prompting for data-centric tasks Prompting LLMs has been quite effective in practice across domains [137, 164, 111]. In this paper, we ask the question: *how does data context impact code generation for data-centric tasks?* Previous works have explored prompting with data:

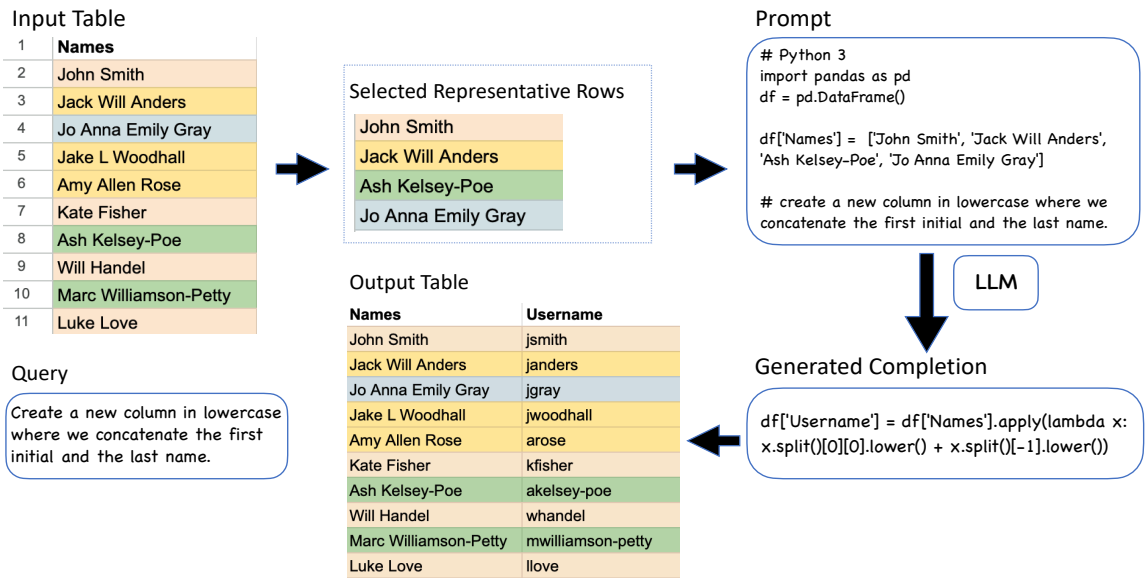


Figure 5.1: An overview of our *cluster-then-select* prompting technique. The input is a data table and natural language query. The rows in the data table are first clustered based on their syntactic structure (in this case the name format). We depict different clusters using distinct colors. The most representative rows are then selected from each cluster to create a prompt to pass to the model. Finally, the generated completion is used to create an output column.

[87] provide both input and expected output tables (which might not be available in a realistic setting). [68] prompt with transformed tables after filtering out rows that are not relevant, for their question-answering tasks. [182] decompose a huge table into a smaller one, and convert a question into simpler sub-questions for tabular question-answering tasks. [83] serialize data tables into a textual representation for tabular classification tasks. These works prompt LLMs for data analysis, classification and wrangling tasks (in-place data transformations) whereas we focus on multi-step data manipulation. We propose a new *cluster-then-select* prompting technique that clusters the input data and adds representative rows to the prompt.

5.3 The SOFSET Dataset

We collect a new dataset fashioned from real-world data-centric tasks from StackOverflow (SOFSET). We sample tasks deterministically from the highest rated posts with the tag "ExcelFor-

mulas" in StackOverflow (as of March 2022). These tasks are representative of real problems spreadsheet users face frequently since they correspond to the highest-rated posts. We manually check that the posts are genuine tasks and also remove post identifiers for anonymization. This gives us a total of 201 tasks.

5.3.1 Dataset Annotation

Each datapoint in our dataset is annotated with a concise textual query, a data input (column-major-flat table), an expected correct output (extra columns), a pandas solution and metadata. We manually write the textual queries, summarising the original verbose StackOverflow question. Each query is annotated and verified by at least three internal annotators. For the data input, we use the table from the original StackOverflow post (if present), and manually add extra rows and corner cases until we have at least 10 rows. Since the natural language query and tabular data are not verbatim copies from StackOverflow and we have a different target language for generation (Pandas instead of Excel Formulas), the evaluation data should not be present in the training data. We choose Pandas as the target language since LLMs are especially good at generating Python but our methods and dataset are programming-language agnostic.

5.3.2 Dataset Properties

What makes our dataset different from existing ones? First, our dataset consists of complex data-centric tasks with multiple input columns. Python datasets like APPS [85] and HUMANEVAL [33] are not data-centric. Second, our dataset is larger than existing data-centric datasets: JIGSAW [87] and CERT [184]. JIGSAW has 79 unique tasks (median of 7 data rows) and CERT has 100 unique tasks (median of 3 rows). Our dataset has 201 unique tasks, with a median of 10 rows. The SPIDER dataset [183] is a text-to-SQL dataset which focuses on relational query tasks whereas we focus on fine-grained data wrangling and manipulation tasks. Finally, we propose a *taxonomy* of data-centric tasks, classifying them into data-independent (IND), data-dependent (DEP), and external-dependent (EXT), based on the data required to produce a

solution.

Data-independent tasks These tasks can be solved using the query alone without any data access. An example is the query "create a new column that includes only the first 5 characters from Filename".

Data-dependent tasks These tasks cannot be solved using the query alone: the model needs access to the input table. For example, the query "create a new column with the number of days between the two date columns" requires data access to identify the correct column names and date format, both absent from the query.

External-dependent tasks These tasks can only be solved with external world knowledge in addition to data access. The query "create a new column that counts how many US holidays are between the dates in Start Date and End Date", requires the model to know about US holidays.

Following this taxonomy, SOFSET consists of 126 IND tasks, 44 DEP tasks and 31 EXT tasks. These tasks span diverse domains including string manipulation, date and time, math, address, and complex conditionals among others.

5.3.3 Cluster-then-select prompting technique

To solve tasks on large tables, we propose a *cluster-then-select* technique which prompts the model with a representative sample of the input data. In order to capture the syntactic variation in the input data, we rely on an existing tool [125], which takes as input a set of strings and synthesizes a small set of regular expressions (regexes), such that each input string matches one of the regexes. In our example in Fig. 5.1, it would synthesize separate regexes for rows with zero, one and two middle names and hyphenated last name. Names like "John Smith" would belong to the zero middle name cluster and "Jack Will Andres" and "Jo Anna Emily Gray" belong to the clusters with one and two middle names resp. Also, the name "Ash Kelsey-Poe" would belong to the cluster with hyphenated last names. These regexes are then used to cluster the input strings, and we select some number of rows from each cluster. In Fig. 5.1, we pick one row from each of the four distinct clusters (depicted with different colors).

If the input table only has one column, selecting n representative rows based on the clustering results is trivial: simply pick one row each from the top-n most populous clusters. In cases where the input contains more than one column, they may be clustered differently. We then select n rows that together cover as many strings as possible across all the columns. We frame this as a *weighted maximal coverage problem* [2], which can be solved approximately in a greedy manner. In each iteration, the algorithm selects the row whose elements maximize cluster coverage.

Kaggle-augmented dataset In order to evaluate our *cluster-then-select* technique on larger datasets, we expand the 44 data-dependent tasks by adding more rows from open-source Kaggle datasets [1], bringing the total to 1000 rows. We first identify the data domains in the original SOFSET rows (such as names, numbers, address, date, time etc) and then source comparable open-source Kaggle datasets of the same domain. We further post-process the Kaggle data to maintain the original rows format and ensure that the augmented data is coherent. This introduces greater variation in the original data which increases the number of data clusters. 62% of our DEP tasks have at least two clusters and we have tasks with up to ten clusters. Since the Kaggle data is post-processed and is not tied to the task query in any way, it is unlikely to bias the LLM evaluation by being part of the training data. This larger dataset allows for a thorough evaluation, better mirroring real-world conditions.

5.4 Evaluation of data-centric tasks

We perform an analysis of the role of data on model performance in data-centric tasks. We first use the original SOFSET dataset to examine three data regimes with increasing amounts of data: (a) no-data (b) first-row and (c) ten-rows and also the taxonomy of task classes of increasing difficulty in terms of data required: IND, DEP and EXT. We then use Kaggle-augmented DEP tasks to compare our *cluster-then-select* technique (which selects representative rows from the top-n most dense clusters) against a random baseline (which selects random rows from the input

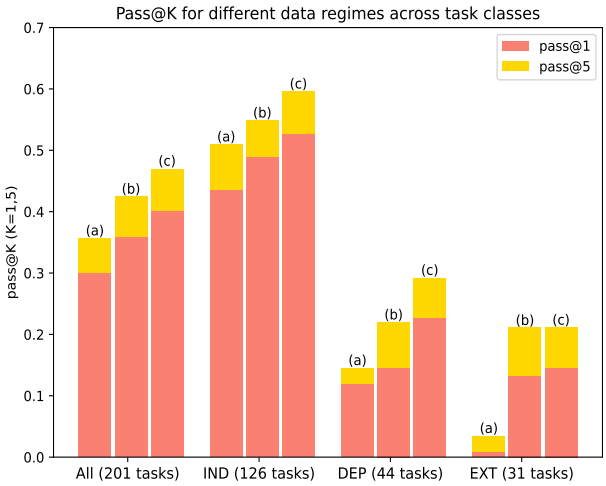


Figure 5.2: pass@k with (a) no-data, (b) first-row, and (c) ten-rows passed to the model. The leftmost group of bars represent pass@k with all classes followed by separate pass@k for IND, DEP and EXT tasks.

table). For each data setting, we construct a prompt which contains the task query and selected rows as a pandas dataframe to generate code from GPT-4 as shown in Fig. 5.1. Correctness is reported based on whether the code produces the expected output in terms of pass@k, the probability that at least one of k samples of generated code produces the correct output [33]. We report all results using GPT-4 with a temperature of 0.5 and the generated completions are evaluated on all rows in the input table. The SOFSET dataset, all the evaluation results and our prototype tool can be found online.[1]

Does model performance vary with the amount of data passed for different task classes? Fig. 5.2 shows the impact of the amount of data on LLM performance, first for the entire dataset and then split by task classes. We see a larger drop in performance with reduced (and no) data on DEP (and EXT) tasks compared to IND tasks. Specifically, the performance gap (pass@5) between first-row and no-data regimes is larger for the DEP and EXT classes (33.8% and 83.5% resp) compared to only 7.1% for IND tasks. The fact that there is any performance drop for IND tasks indicates that having data helps the model even when the problem can be solved independently of data. In the absence of data, almost no EXT task is solved (pass@1) but performance improves when a single row is passed.

[1]https://github.com/microsoft/CodeXData

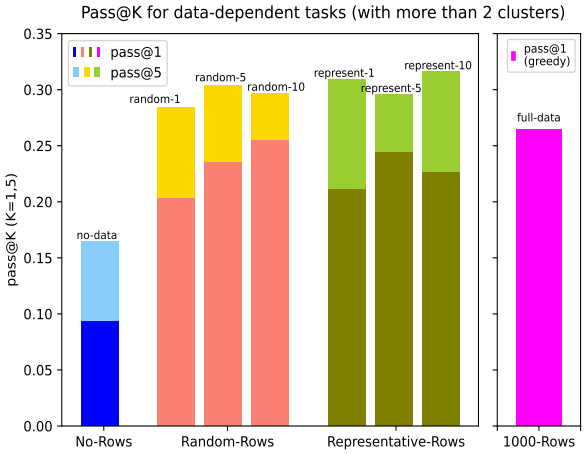


Figure 5.3: pass@k for 39% (17/44) DEP tasks (with more than two clusters) with no-data, random selection (random-n), representative selection (represent-n) and pass@1 with greedy sampling for full-data (1000 rows).

Is our cluster-then-select technique effective on larger input tables? We evaluate our *cluster-then-select* technique on Kaggle-augmented DEP tasks (with 1000 rows) since we expect to see the benefit of our approach more clearly on tasks dependent on data. In order to do so, we compare our representative selection strategy against *random* selection where the rows are randomly selected from the input table. Among DEP tasks, we further focus on 17 (out of 44) that have input columns with at least three clusters, since with two clusters or fewer we do not expect to see much difference between the representative and random samples. We also evaluate against two baselines: no-data (0 rows) and full-data (1000 rows). We run the random selection experiments five times.

Fig. 5.3 shows that the model performs best with 10 most representative rows added to the prompt (pass@5 = 0.32 for represent-10). Representative selection performs better than random selection for the same number of rows. Specifically, represent-1 and represent-10 outperform random-1 and random-10 by 8% and 6% resp. In addition, random selection has *high variance*, especially for a small number of rows (*e.g.* pass@1 for random-1 varies from 0.20 to 0.31 across the five runs), which is not surprising, since the random strategy might select rows from different clusters or from the same one. Thus, while random selection gives comparable results on average,

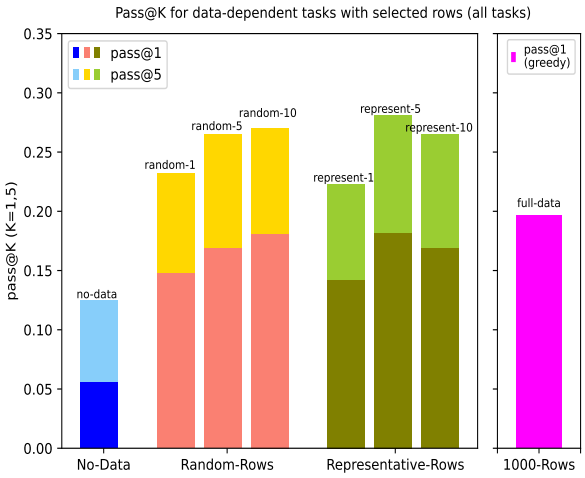


Figure 5.4: pass@k for all DEP tasks with no-data, and n=1, 5 and 10 rows passed to the model, using random (random-n), representative selection (represent-n). The completions are evaluated on 1000 rows.

our cluster-then-select technique offers a more consistent approach to provide the model a representative sample of the data. Further, the low pass@k for the no-data baseline suggests that our dataset was not part of the training data, as then the model would likely perform well even without data input. We note that while we evaluate on 1000 rows, the same cluster-then-select technique could easily scale to datasets with over 100K rows without much overhead. We also present the evaluation results on all the 44 DEP tasks in Fig. 5.4. We see that represent-5 has the highest pass@k for both k = 1 and 5. Since these results include problems with fewer than three clusters, selection of even 5 representative rows boosts performance. Notably, represent-5 also outperforms random-10.

Does the position of data rows in the prompt also affect performance? For the full-data baseline, we used a longer-context version of GPT-4 (32k) with temperature 0 (*greedy selection to eliminate variance in generations*) for the DEP tasks. The right side of Fig. 5.3 shows pass@1 for this setting with ten runs: we permute the 1000 rows in the dataframe ten times, in order to measure the sensitivity of the model to row positioning. We observe a high variance in pass@1 values, ranging from 0.20 to 0.32 with an average of 0.26. This shows that the position of rows in the dataframe influences completion quality, which aligns with previous

findings about positional biases in prompts [110]. Surprisingly, the full-data setting (irrespective of row ordering) performs worse than selecting one random row *in some cases* (pass@1 for one random row ranges from 0.12 to 0.27 with an average of 0.20). Note that we only report pass@1 results for the full-data (1000 rows) setting.[2]

5.5 Conclusion and Future Work

Our work highlights the importance of data for code generation on data-centric tasks and proposes a new dataset for evaluation of data-centric tasks. We show that providing even one data row to the model boosts performance compared to a no-data baseline. Since providing the entire input table is often infeasible, we propose a *cluster-then-select* prompting technique that selects representative rows from the data to be added to the prompt. While randomly selecting rows also performs well, for data with a high degree of syntactic variation, it is more beneficial to add representative rows to the prompt. For future work, handling a broader problem space (*e.g.*, multi-table inputs, hierarchical table inputs) raises interesting challenges.

5.6 Limitations

We discuss the limitations of our work in terms of the SOFSET dataset, the *cluster-then-select* technique and the models used for evaluation. Although starting from actual user-specified problems gives our results greater alignment with real spreadsheet user problems, the form that such queries take pose some potential limitations to our analysis. Users usually only show relevant columns of data in their queries when in actuality there might be many more unrelated columns in real spreadsheets. We have seen promising results applying LLMs to data tables with columns that are extraneous to the query but we do not perform a rigorous evaluation of the same. Furthermore, since we have collected only English queries from StackOverflow, our results may not generalize to other languages.

[2] CODELLAMA results are Fig. 5.6, Fig. 5.7, Fig. 5.8.

Our cluster-then-select prompting technique is based on the regular expression synthesis algorithm from [125]. Given that the clusters for the input data columns are defined by the specificity of this regex synthesis, using a different clustering algorithm could potentially result in a different set of clusters. Finally, since we draw our conclusions from the generations produced by GPT-4, future models might invalidate our conclusions. Furthermore access to models such as GPT-4 cannot be taken for granted and the costs of running our evaluation are considerable. Even open source models like CODELLAMA require GPU resources for evaluation.

5.7 Broader Research Impact

To the best of our knowledge, research on prompting large language models to solve data-centric tasks with tabular data is infrequent, despite the considerable importance of such scenarios. Solving the problem of how to help LLM reason over large amounts of data is essential to the future of assisted decision making. Generating multi-step programs that require reasoning is the beginning of this journey and to make progress the community needs challenging real-world datasets to evaluate on. By releasing our new dataset, sharing the analysis results of our experiments and releasing our prototype tool[3], we offer valuable benchmarks and a baseline to the wider research community which promises to encourage further exploration.

5.8 Ethics Statement

There are broad ethical impacts resulting from the creation of AI models that attempt to generate code solutions from natural language descriptions and these are discussed in detail in previous papers including CODEX [33], and PALM [38]. These impacts include over-reliance, misalignment between what the user expressed and what they intended, potential for bias and under/over representation in the model results, economic impacts, the potential for privacy and security risks, and even environmental considerations. All of these considerations also apply

[3]Details discussed in Sec. 5.9 and Sec. 5.10.

to the work in this paper. Our focus is to highlight how the presence of data improves the performance of these models but it is important to note that the quality of the data used in the prompt will impact whether the resulting generation exhibits bias, exposes private data, etc. We explore the overall impact of providing data as part of the prompt but do not conduct a more focused analysis of determining how bias in the prompt data might influence the resulting code generation, a task we leave for future work.

There is the question of the sources of data and of consent to use the data in the manner exhibited in this paper. We have reviewed each of the datasets we have included in this paper to ensure that our use is compatible with the intent of the authors and publishers. Our datasets have also been reviewed by our institution's ethics board to review that this is an ethical use.

This paper does not directly contribute to a tool built on the assumed capabilities of language models to understand data, but nonetheless, it is motivated by their potential applications in such tools. These tools may be deployed in many data applications such as databases, spreadsheets, and business intelligence applications. Depending on the audience of the tool, various interaction design concerns arise. Explainability of the model is a key consideration, and the tool should offer decision support to evaluate mispredictions and potential next steps [139]. Previous research of non-experts using inference driven tools for data manipulation has shown the importance of tool design in the critical appreciation of the model and its limitations, and in the potential cost of errors [175, 141]. As an exploratory paper without a concrete application, we do not encounter these issues, but the project has nonetheless been reviewed by our institution's ethics board.

5.9 Our Prototype Tool

The high-level workflow of our tool is depicted in Fig. 5.5 and formalized in Algorithm 3. The tool takes as input a query Q expressed in natural language, an input table T as a Pandas dataframe, and the target cardinality k of distinct completions to generate. We set a limit k_{max}

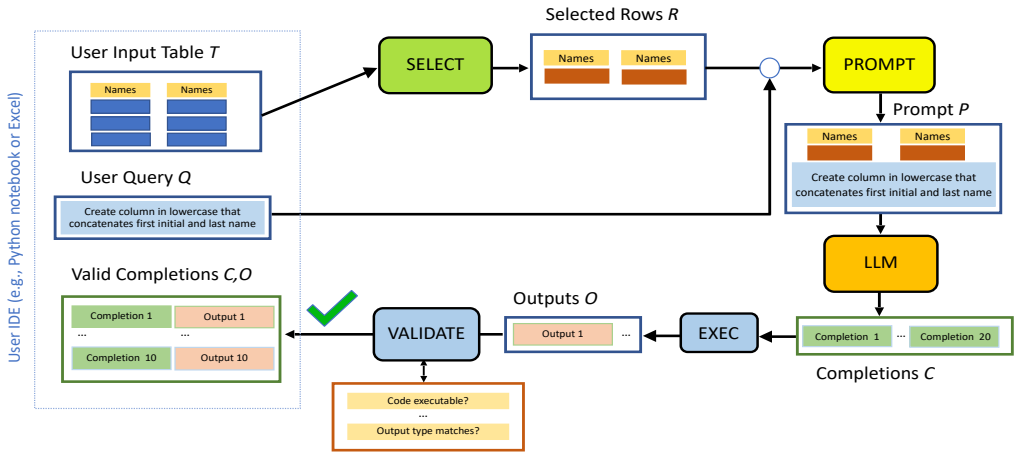


Figure 5.5: Our tool transforms an input table and a query into a list of valid completions. The input data is used to extract the selected rows R. The resulting rows and query are used to construct a prompt which is fed to a code synthesis LLM, such as GPT-4 or CODELLAMA, generating multiple possible completions. The outputs of these completions are then validated and the first k valid completions (along with the outputs) are returned.

on the number of calls to LLM ($k_{max} = 8k$). For our running example, k is 1, Q is "create a new column in lowercase that concatenates the first initial and the last name.", and T is `Data({"Names":["John Smith", "Jack Will Anders", ...]})`. At a high-level, the algorithm first clusters the data in T based on automatically synthesized regular expressions and stores them in a map M (line 2). It then extracts *representative rows* of the table using SELECT (line 3); combines the query Q and the rows R to create a prompt P using PROMPT (line 4); and then queries LLM repeatedly using this prompt until the target completions are reached or we exceed the budget of calls (lines 7-12). Each completion c is executed on the input table (line 9) using an EXEC procedure, and if the completion is new and its output o satisfies a VALIDATE procedure, the two are accumulated in C and O which are then returned. We describe each of the procedures in detail below.

CLUSTER This procedure clusters the rows in the input table T based on their syntactic structure. To capture the syntactic variation among input rows, we rely on an existing tool [125], which takes as input a set of strings and synthesizes a set of regular expressions (regexes) from a

Algorithm 3. Inference Algorithm

Input: Explicit: query Q, input talble T, cardinality k. Implicit: completion limit k_{max} (with $k \leq k_{max}$), number n of rows to be selected.

Output: Pair of lists (C, O), with $|C| = |O| \leq k$, of unique completions and their corresponding outputs.

```
1:  procedure INFER(Q, T, k)
2:      M ← CLUSTER(T)                          ▷ cluster input rows
3:      R ← SELECT(T, n, M)                     ▷ select n representative rows
4:      P ← PROMPT(Q, R)                        ▷ prompt creation
5:      B, C, O ← k_max, [], []                 ▷ initialize budget, caches
6:      while B > 0 ∧ |C| < k  do
7:          c ← LLM(P)                          ▷ sample completion
8:          B ← B − 1                           ▷ decrement budget
9:          o ← EXEC(c, T)                      ▷ execute against table T
10:         if VALIDATE(o) ∧ (c ∉ C) then
11:             C ← C + [c]                     ▷ append completion to C
12:             O ← O + [o]                     ▷ append output to O
13:     return (C, O)
```

restricted class, such that each input string matches one of the regexes. In our example, the tool synthesizes four regexes: `[A-Z][a-z]+[\\s] [A-Z][a-z]+`, for rows with no middle name like "John Smith", and similar regexes for rows with dashed last names like "Ashley Kelsey-Poe", and one or more middle names. These regexes are then used to cluster input strings.

SELECT The SELECT procedure selects the top-n most representative rows from the input table. We frame the selection of most representative rows as a *weighted maximal coverage problem*— a well-known NP-complete problem [2] that can be solved approximately using the greedy algorithm in Algorithm 4. The algorithm takes as input the table T, a map M from the rows of the table to the set of clusters covered by the element in each column of the row. It also takes as input the row budget n. The algorithm iterates over all rows in T not already in R (line 3) and in each iteration selects the row whose elements maximize the size of clusters covered (line 4), adding this row to R.

PROMPT The prompt creation procedure PROMPT creates a textual prompt by concatenating the NL query and the representative rows R which are in form of a Pandas dataframe. An example prompt is in Appendix 5.10.3.

LLM The completion procedure LLM queries GPT-4 (or another code-generating model),

Algorithm 4. Rows Coverage Algorithm SELECT

```
1: procedure SELECT(T, n, M)
2:     while |R| < n do
3:         for r ∈ T_r ∧ r ∉ R do
4:             BEST ← argmax(∑{ |c_i| | c_i ∈ M[r] })
                                          ▷ greedily increase coverage
5:         R ← R ∪ BEST
       return R
```

passing the prompt P and also the predefined stop sequences. We use stop sequences that we have found to allow the LLM to generate at least one solution while typically not using the entire token budget. Note that the LLM needs to produce multiple completions, because it will filter out invalid completions. A naive approach would be to request a single completion, validate it, and repeat the process until k distinct valid completions are obtained; this, however, requires sending the prompt to the LLM every time, which incurs a monetary cost. An alternative approach is to *batch* the completions, *i.e.* request some number b of completions in parallel; if the batch size b is too large, however, this also incurs unnecessary cost, since we are requesting more output tokens than we need. Details in Appendix 5.10.4.

EXEC The procedure EXEC turns each LLM completion into a stand-alone executable program and runs it to obtain the final output o. There are two main challenges to be addressed in this step. First, LLM completions do not have a consistent way of identifying the final output: for example, the last line of the completion might be an expression that computes the output, or an assignment to a `result` variable, or a print statement. So our tool uses a predefined set of rewrite rules, which we developed by analyzing the patterns in completions. The second challenge is that executing arbitrary LLM-generated code poses a security risk; for this reason, we execute completions in a sandbox. Further details are available in Appendix 5.10.5.

VALIDATE The procedure VALIDATE checks that the output value o is a dataframe with the right dimensions. The completions that executed without runtime errors during EXEC and passed the output validation are deemed *valid*. Further details are available in Appendix 5.10.6.

5.10 Experimental Details

5.10.1 CODELLAMA Results

We do a performance comparison for no-data, first-row and full-data regimes and the different selection strategies with CODELLAMA as the LLM. The results with CODELLAMA are presented in Fig. 5.6, Fig. 5.7 and Fig. 5.8.

5.10.2 Evaluation Metrics

The probability that at least one of k inferred outputs is correct is called pass@k [33]. More formally, pass@k is the probability that with a sample of k code completions, at least one is correct. To measure this probability empirically for each datapoint, we compute up to m valid programs by sampling from the LLM (GPT-4 or CODELLAMA). We count the number s of correct completions, and hence compute an estimate of pass@k as $1 - \binom{m-s}{k} / \binom{m}{k}$ [33]. By computing $m > k$ completions the estimate has lower variance than by simply computing k completions. Each pass@k on a whole dataset is the average of pass@k over all its datapoints. All evaluation results are averaged over tasks, computing m valid completions to estimate pass@k or pass@$k(X\%)$. In practice, we set $m = 20 * k$ when we report results for $k = 1$ or $k = 5$.

5.10.3 Prompt Template

For each task, we generate prompts according to the data regimes and selection strategies as described above. An example prompt for the query "Create a new column with the difference in hours, minutes and seconds between the two timestamps in the format HH:MM:SS" with one row selected:

```
import pandas as pd
df = pd.DataFrame()
df[‘Start’] = [‘2/22/2015 1:06:20 PM’]
df[‘End’] = [‘2/23/2015 3:08:20 PM’]
```

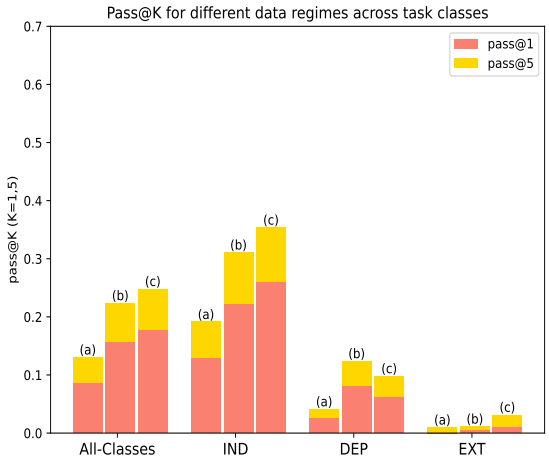


Figure 5.6: pass@k (for CODELLAMA) with (a) no-data, (b) first-row, and (c) full-data (10 rows) passed to the model. The leftmost group of bars represent pass@k with all classes followed by separate pass@k for IND, DEP and EXT tasks. Smaller models have a huge performance drop. But the trend of performance improving with the amount of data passed to the model is seen.

```
#Create a new column with the difference in hours, minutes, and seconds
   between the two timestamps in the format HH:MM:SS
```

Listing 5.1. Example of a prompt

5.10.4 Generation of Completions

Parallelization. For efficiency, we request multiple completions from GPT-4 per iteration. To try to minimize both inference time and the load on OpenAI's servers, we adapt the batch size to an estimate of the probability that the next completion is valid. The batch size used in each iteration is $n = \min(\lceil r/p \rceil, B, L)$, where $r = k - |C|$ is the number of valid completions still to obtain, B is the remaining completion budget, and L is a parallelization limit enforced by the GPT-4 API. The probability estimate p is updated after each iteration by counting the number of valid and invalid completions in that iteration's batch. Since pass@k is calculated only from valid completions, it is not influenced by either parallelization or batch size adaptation.

Stop sequences. The most effective stop sequence we found that allows GPT-4 to generate at least one solution while not usually using the entire token budget is a blank line followed by a

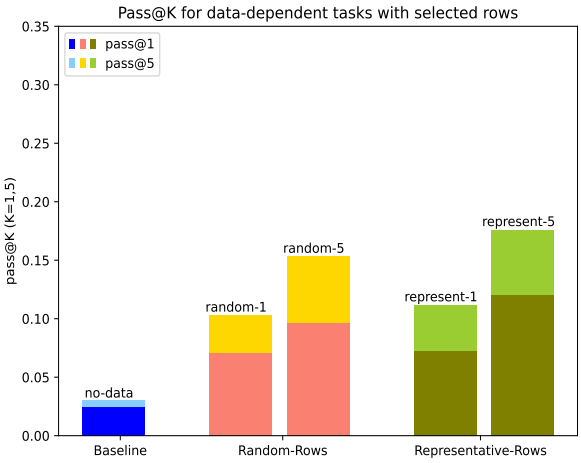


Figure 5.7: pass@k (for CODELLAMA) for all 44 DEP tasks with no-data, and n=1 and 5 rows passed to the model, using random (random-n) selection, representative selection (represent-n) and full-data (1000 rows). Completions are evaluated on 1000 rows.

line comment; i.e. `\n#`. Further, to keep GPT-4 from generating what appears to be the rest of a forum post after a code snippet, we also use the stop sequence `</code>`.

Completion cleanup. Since GPT-4's training data likely contains forum posts, some completions would raise `SyntaxError` exceptions when executed due to formatting artifacts, and therefore be invalid. Instead, to make the most of the completion budget, we replace formatting artifacts *i.e.* we replace HTML escape sequences such as `&lt;` and `&quot;` with Python operators and delimiters. Cleanup also removes unnecessary whitespace, blank lines, comments, and truncates completions at `\n#` when it appears after executable code.

5.10.5 Execution of Completions

Rewriting. Completions returned by GPT-4 do not clearly indicate which variables or expressions are intended to be the answer to a query. This must be inferred from the shape of the code. We found that an effective way to identify and expose the likely answer is to search backwards to find the last unindented (i.e. top-level) statement that has one of a few forms, and rewrite the completion so that its last statement is an assignment to a fresh identifier `var_out`. The statement forms and rewrites are

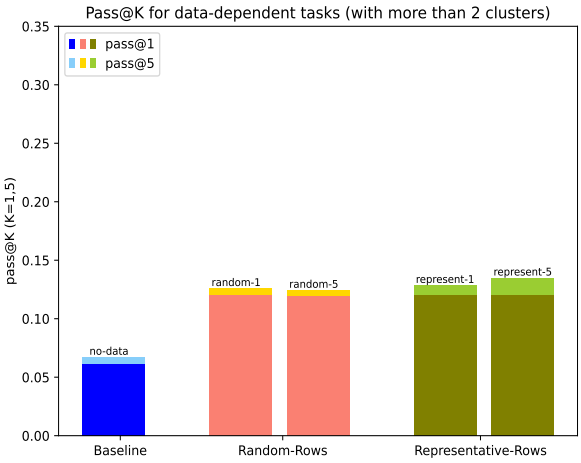


Figure 5.8: pass@k (for CODELLAMA) for 17 out of 44 DEP tasks (more than two clusters) with no-data, random selection (random-n) and representative selection (represent-n). Completions are evaluated on 1000 rows.

- `var = expr`: append the statement `var_out = var` to the completion.
- `var[expr_i] = expr`: append the statement `var_out = var` to the completion
- `print(expr, ...)`: replace this statement and the rest of the completion with `var_out = expr`
- `expr`: replace this statement and the rest of the completion with `var_out = expr`

Rewriting also inserts `import` statements for common libraries (e.g. `import numpy as np`). The rewritten completion is appended to the code that defines the input dataframe to create a complete program. The program and output variable `var_out` are sent to a sandbox for execution.

Sandboxing. Because of security risks inherent in running the LLM-generated code, we run completed programs in a sandbox. Our sandbox is a JavaScript web service that runs Python programs in Pyodide [44], a Python distribution for WebAssembly. While Python programs running in Pyodide have access to the host's network resources, they at least are isolated from other host resources including its filesystem, offering some level of protection from malicious or accidentally harmful completions. After running the code, the sandbox returns the value of `var_out`.

5.10.6 Validation of Completions

For a completion to be considered a correct solution in the calculation of pass@k, its actual output must match the expected output. Matching cannot be the same as equality and still conform to a reasonable notion of correctness; for example, the natural breakdown of a solution might generate intermediate columns in the actual output that are not in the expected output. The actual output is allowed to vary from the expected output in the following ways and still match the expected output:

- Extra columns
- Different column order
- Different column headers
- Number expected; actual is a number within small relative error (default 0.01)
- Number expected; actual is a string that parses as a number within small relative error
- Boolean expected; actual is number 0 or 1
- Boolean expected; actual is a string that represents a truth value
- String expected; actual is a string that differs only in case

5.11 Acknowledgements

Chapter 5, in full, is a reprint of the material as it appears in Solving Data-centric Tasks using Large Language Models. Barke, Shraddha; Poelitz, Christian; Negreanu, Carina; Zorn, Benjamin; Cambronero, José; Gordon, Andrew; Le, Vu; Nouri, Elnaz; Polikarpova, Nadia; Sarkar, Advait; Slininger, Brian; Toronto, Neil; Williams, Jack. Findings of the Association for Computational Linguistics: NAACL 2024. The dissertation author was a primary investigator and author of this paper.

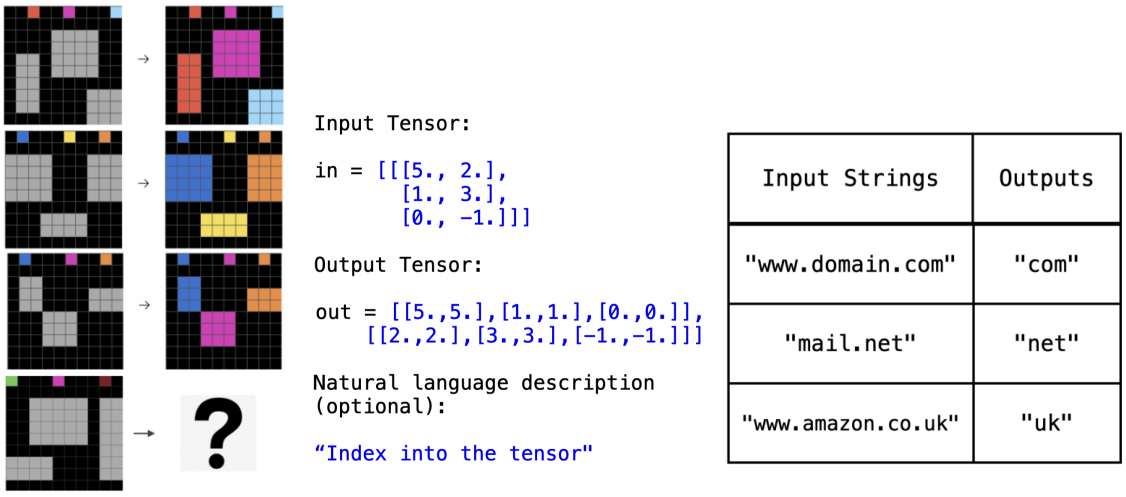
Chapter 6

HySynth: Context-Free LLM Approximation for Guiding Program Synthesis

6.1 Introduction

Large language models (LLMs) demonstrate impressive capabilities in various domains, but they continue to struggle with tasks that require precision—e.g. structured prediction, reasoning, counting, or data transformation—when direct task examples are not prevalent in their training data [156, 180, 22, 18, 92, 160, 115]. As one example, consider the *Abstraction and Reasoning Corpus* (ARC) [36], which was designed as a benchmark for human-like structured reasoning. ARC tasks are grid-based puzzles, such as one depicted in Fig. 6.1a. This puzzle consists of three training examples, which are pairs of input and output grids; the goal is to infer the transformation that maps the input to the output, and then apply this transformation to the test grid. The ARC benchmark's emphasis on generalization and few-shot learning has rendered it challenging to solve with purely machine learning techniques: state-of-the-art generative models like GPT-4 hardly solve more than 10% of the tasks in the dataset when asked to predict the test output, even with the help of advanced prompting techniques [101].

In fact, the leading entries in the ARC Kaggle competition [4] tackle this task using *Programming-by-Example* (PBE): instead of predicting the output directly, they search for a program that captures the transformation occurring in the input-output examples. For example,



(a) ARC **(b)** TENSOR **(c)** STRING

Figure 6.1: Example problems from the three domains we evaluate HYSYNTH on: grid-based puzzles (ARC), tensor manipulation (TENSOR), and string manipulation (STRING).

the transformation in Fig. 6.1a might be represented as the following program:

```
if color_of(self) = GREY ∧ is_neighbor(self, other) ∧ size_of
    (other) = MIN
                        then update_color(color_of(other))   (6.1)
```

This particular program is written in a *domain-specific language* (DSL) inspired by the ARGA tool [179]. It consists of a single *rule* of the form `if` *filter* `then` *transform*, which is applied to each object in the grid simultaneously; if the filter holds for the focus object `self` and another object `other`, then `self` undergoes the transform. In this case, the rule says that any grey object that has a neighbor of the grid's minimum size (here, a single pixel) should be colored with the color of that neighbor.

Beyond grid puzzles, PBE is a general paradigm for structured reasoning and data transformation tasks: for example, it can help spreadsheet users with systematic string manipulation [74], and help programmers use unfamiliar APIs [57, 54, 144]; Fig. 6.1 shows example PBE tasks from three domains.

Challenge: Harnessing the Power of LLMs for PBE. How can we automatically discover programs from the input-output examples like those shown in Fig. 6.1? The traditional *program synthesis* approach is based on combinatorial search [159, 5, 124, 15, 136], which works well for small programs and restrictive DSLs, but becomes infeasible as the program size and the DSL complexity grow. At the other end of the spectrum, purely *neural* approaches [41, 174] use a neural model to predict the program from input-output examples; unfortunately, even state-of-art LLMs like GPT-4o [123] struggle to predict an entire program in an unfamiliar DSL: when we asked GPT-4o to generate 10 programs for the running example above, none of them were entirely correct.[1]

In the past, the limitations of both program synthesis and neural techniques have motivated a hybrid approach, where combinatorial search is *guided* by a learned probabilistic model [19, 94, 102, 122, 144, 145]. Existing hybrid techniques, however, use domain-specific models trained on datasets of similar PBE tasks, which limits their generalization to new domains. With the advent of LLMs, can we now use a single pre-trained model to guide program synthesis across a wide range of domains?

Interestingly, there is some tension in the hybrid approach between the efficiency of the search algorithm and the power of the model: a search algorithm is efficient when it *factorizes the search space* (*i.e.*, merges many search states into one), which often makes it incompatible with a powerful model that requires a lot of context to make a prediction. Specifically, one of the most widely used program synthesis techniques is *bottom-up search* [5, 159, 21, 144, 106], which is a dynamic programming algorithm, whose efficiency relies on reusing the work of constructing and evaluating subprograms in many different contexts. This essentially precludes using models with unlimited left-to-right context—like LLMs–to guide bottom-up search.

Our Solution: Context-Free LLM Approximation. To bridge this gap and harness the power of LLMs to guide bottom-up search, we propose to approximate the LLM's conditional output distribution *for a given task* with a context-free surrogate model. Recent work in

[1] A detailed analysis of GPT-4o's performance on this task is provided in Sec. 6.7.

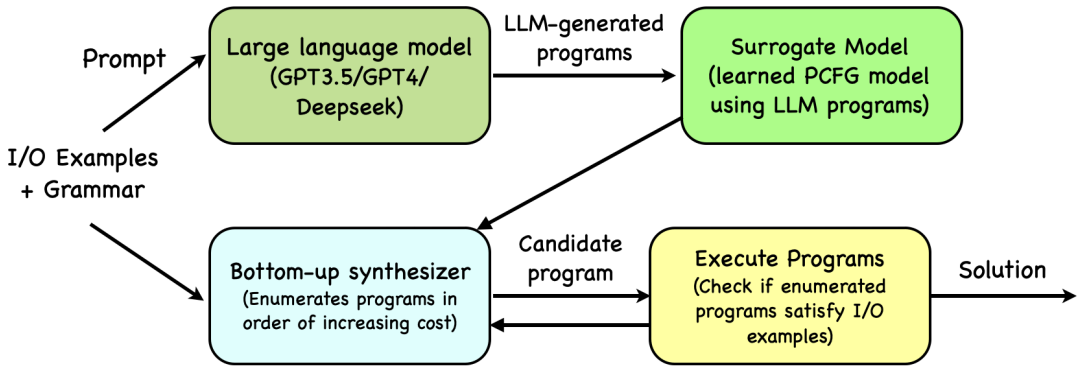


Figure 6.2: An overview of the hybrid program synthesis technique that uses a context-free LLM approximation. Programs generated by an LLM are used to learn a PCFG, which guides a bottom-up synthesizer to generate programs until a solution is found.

NLP [185] has found that a Hidden Markov Model (HMM) trained to match an LLM can be used as an efficient surrogate in style-controlled language generation. We extend this idea to program synthesis, replacing the HMM with a *probabilistic context-free grammar* (PCFG). The benefits of using a PCFG are twofold:

(1) PCFGs are context-free, which makes them compatible with bottom-up search for PBE [21, 144], and

(2) while a context-free model may make a poor approximation to an LLM's full joint, in a PBE setting it is able to reasonably approximate an LLM's conditional distribution over output programs *for a given prompt*.

The overview of our approach is shown in Fig. 6.2.

Evaluation. We implemented this technique in a tool HYSYNTH[2] and evaluated it on 299 PBE tasks from three domains: ARC grid-based puzzles [36], tensor manipulation tasks from TFCODER [144], and string manipulation tasks from the SYGUS benchmark [12], which are inspired by spreadsheet use cases. Example problems from these domains are shown in Fig. 6.1. Our evaluation shows that HYSYNTH outperforms both unguided search and LLMs alone, solving 58% of the tasks overall, compared to 40% for unguided search and 2% for LLMs

[2]The name stands for "HYbrid SYNTHesis" and is pronounced like the flower "hyacinth".

without search. Our tool also outperforms baseline program synthesizers for these domains—ARGA, TFCODER, and PROBE, respectively; importantly, in the TENSOR domain, the guidance from the LLM not only speeds up the search, but also frees the user from having to explicitly provide any non-standard *constants* that the solution might use, thereby significantly improving the usability of the tool.

Contributions. In summary, this paper makes the following contributions:

1. We propose a hybrid program synthesis approach that integrates LLMs with efficient bottom-up search via a task-specific context-free approximation.

2. We implement this approach in a tool HYSYNTH and instantiate it on three domains: grid-based puzzles (ARC), tensor manipulation (TENSOR), and string manipulation (STRING). While the latter two domains reuse off-the-shelf bottom-up synthesizers, for ARC we implement a custom synthesizer that uses a divide-and-conquer strategy [13] to leverage the structure of the rule-based DSL to further speed up the search.

3. We evaluate HYSYNTH on the three domains and show that it outperforms both the LLM alone and existing baseline synthesizers, which are not guided by LLMs.

6.2 Background

6.2.1 Programming-By-Example

Programming by Example (PBE) [76] is the task of synthesizing programs that satisfy a given set of input-output examples. To restrict the program space, the programs are typically drawn from a *domain-specific language* (DSL), which is specified by a *context-free grammar* and an *evaluation function*. This section provides a formal definition of these concepts.

Context-Free Grammars. A *context-free grammar* (CFG) is a tuple $\mathscr{G} = (\mathscr{N}, \Sigma, \mathscr{S}, \mathscr{R})$, where $\mathscr{N}$ is a set of non-terminal symbols, Σ is a set of terminal symbols, $\mathscr{S} \in \mathscr{N}$ denotes the starting non-terminal, and $\mathscr{R}$ is the set of production rules. An example CFG is shown in

Rule →	`if` *Filter* `then` *Transform*	*Color* →	`color_of(`*Obj*`)` | `GREY`...
Filter →	*Atom* | `not` *Atom* | *Atom* ∧ *Filter* | ...	*Size* →	`size_of(`*Obj*`)` | `MIN`...
Atom →	*Color* $=_c$ *Color* | *Size* $=_s$ *Size* | ...	*Dir* →	`dir_of(`*Obj*`)` | `UP`...
Transform →	`update_color(`*Color*`)` | `move(`*Dir*`)` | ...	*Obj* →	`self` | `x` | `y` | ...

Figure 6.3: A fragment from the context-free grammar of our ARC DSL.

Fig. 6.3. We denote with $\mathscr{R}(\mathrm{N})$ the set of all rules $\mathrm{R} \in \mathscr{R}$ whose left-hand side is N. A grammar $\mathscr{G}$ defines a (leftmost) *single-step derivation* relation on sequences of symbols: $s\mathrm{N}\alpha \Rightarrow s\beta\alpha$ if $\mathrm{N} \to \beta \in \mathscr{R}$, where $s \in \Sigma^*$ and $\alpha, \beta \in (\mathscr{N} \cup \Sigma)^*$. The transitive closure of this relation $\Rightarrow^*$ is called (leftmost) *derivation*.

Programs. A *program* $P \in \Sigma^*$ is a terminal sequence derivable from some $\mathrm{N} \in \mathscr{N}$; we call a program *whole* if it is derivable from $\mathscr{S}$. The set of all programs is called the *language* of the grammar $\mathscr{G}$: $\mathscr{L}(\mathscr{G}) = \{s \in \Sigma^* \mid \mathrm{N} \Rightarrow^* s\}$. The *trace* of a program $\mathrm{tr}(P)$ is the sequence of production rules $\mathrm{R}_1, \ldots, \mathrm{R}_n$ used in its derivation ($\mathrm{N} \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow P$). The *size* of a program $|P|$ is the length of its trace. The semantics of a program P is defined by the evaluation function $[\![P]\!]\colon \mathrm{Val}^* \to \mathrm{Val}$, which maps the values of program variables to its output value.

Problem Statement. A PBE problem is defined by a DSL with a grammar $\mathscr{G}$ and an evaluation function $[\![\cdot]\!]$, as well as a set of input-output examples $\mathscr{E} = \overrightarrow{\langle i, o \rangle}$ where $i \in \mathrm{Val}^*$, $o \in \mathrm{Val}$. A *solution* to the problem is a program $P \in \mathscr{L}(\mathscr{G})$ such that $\forall \langle i, o \rangle \in \mathscr{E}$, $[\![P]\!](i) = o$.

6.2.2 Assigning Costs to Programs

Weighted Context-free Grammar. A *weighted context-free grammar* (WCFG) $\mathscr{G}_w$ is a pair of a CFG $\mathscr{G}$ and a function $w_{\mathbb{R}} : \mathscr{R} \to \mathbb{R}^+$ that maps each production rule $\mathrm{R} \in \mathscr{R}$ to a positive weight. Given a weighted grammar $\mathscr{G}_w$, we can define the *real cost* of a program P as the sum of weights of all the productions in its trace: $\mathrm{rcost}(P) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} w_{\mathbb{R}}(\mathrm{R}_i)$.

For the purposes of search, it is convenient to define a *discrete weight* function $w : \mathscr{R} \to \mathbb{Z}^+$, which rounds weights up to the nearest integer: $w(\mathrm{R}) = \lfloor w_{\mathbb{R}}(\mathrm{R}) \rceil$. The (discrete) *cost* of a program P is defined as the sum of discrete production weights: $\mathrm{cost}(P) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} w(\mathrm{R}_i)$. Note

that because of error accumulation, the discrete cost of a program can differ from its rounded real cost, but the difference can be made arbitrarily small by scaling all the costs by a constant factor $\alpha > 1$.

Probabilistic Context-free Grammar. A popular way to assign weights to production rules is via a *probabilistic context-free grammar* (PCFG). A PCFG $\mathscr{G}_p$ is a pair of a CFG $\mathscr{G}$ and a function $p : \mathscr{R} \to [0,1]$ that maps each production rule $\mathrm{R} \in \mathscr{R}$ to its probability, such that probabilities of all the rules for a given non-terminal $\mathrm{N} \in \mathscr{N}$ sum up to one: $\forall \mathrm{N}. \sum_{\mathrm{R} \in \mathscr{R}(\mathrm{N})} p(\mathrm{R}) = 1$. A PCFG defines a probability distribution on programs: $p(P) = \prod_{\mathrm{R}_i \in \mathrm{tr}(P)} p(\mathrm{R}_i)$.

Given a PCFG $(\mathscr{G}, p)$ we can derive a WCFG $\mathscr{G}_w$ where $w_{\mathbb{R}}(\mathrm{R}) = -\log(p(\mathrm{R}))$; to make sure that all weights are finite and positive, we exclude rules with $p(\mathrm{R}) = 0$ and inline rules with $p(\mathrm{R}) = 1$. In this WCFG, the real cost of a program is related to its probability: $\mathrm{rcost}(P) = -\log(p(P))$.

6.2.3 Bottom-up Search

Bottom-up search is a popular search technique in program synthesis [5, 159, 21, 144, 106], which enumerates programs from the DSL in the order of increasing costs until it finds a program that satisfies the given examples. The search is implemented as a dynamic programming algorithm (see Alg. 5), which maintains a program *bank* B mapping discrete costs to programs of that cost. Starting with an empty bank and current cost level $\textsc{Lvl} = 1$, the search iteratively creates all programs of cost 1, 2, 3, and so on; to create complex programs, the algorithm *reuses* simpler programs already stored in the bank, and combines them using the production rules of the grammar.

For example, consider the CFG in Fig. 6.3, and assume a uniform weight function $w(\cdot) = 1$. Then in the first iteration (cost level 1), the algorithm will enumerate programs consisting of a single literal or variable—*e.g.* `self`, `GREY`, `UP`, *etc*—and store them in $\mathrm{B}[1]$. At cost level 2, it will enumerate unary operators applied to programs stored in $\mathrm{B}[1]$: *e.g.* `color_of(self)`, `move(UP)`, *etc*. More generally, at cost level Lvl, the algorithms

Algorithm 5. Bottom-Up Search Algorithm

Input: Input-output examples $\mathscr{E}$, a WCFG $\mathscr{G}_w = (\mathscr{N}, \Sigma, \mathscr{S}, \mathscr{R}, w)$
Output: A program P consistent with $\mathscr{E}$ or failure ($\perp$)

```
 1: procedure BOTTOM-UP-SEARCH(G_w, E)
 2:     LVL, B, E ← 1, ∅, ∅                          ▷ Initialize state of the search
 3:     while true do
 4:         for P ∈ NEW-PROGRAMS(G_w, LVL, B) do      ▷ For all programs of cost LVL
 5:             EVAL ← [⟨i, [[P]](i)⟩ | ⟨i, o⟩ ∈ E]   ▷ Evaluate on inputs from E
 6:             if (EVAL = E) then
 7:                 return P                          ▷ P fully satisfies E, solution found!
 8:             else if (EVAL ∈ E) then
 9:                 continue                          ▷ P is semantically equivalent to another program in B
10:             B[LVL] ← B[LVL] ∪ {P}                 ▷ Add to the bank, indexed by cost
11:             E ← E ∪ EVAL                          ▷ Cache evaluation result
12:         LVL ← LVL + 1
13:     return ⊥                                      ▷ Cost limit reached
14: procedure NEW-PROGRAMS(G_w, LVL, B)
15:     for R = N → s_0 N_1 s_1 N_2 ... N_k s_k ∈ R do   ▷ R is a production rule with k non-terminals
16:         for (c_1, ..., c_k) ∈ { [1..LVL − 1]^k | Σ c_i = LVL − w(R) } do   ▷ For all subexpression costs
17:             for (P_1, ..., P_k) ∈ { B[c_1] × ... × B[c_k] | ⋀_i N_i ⇒* P_i } do   ▷ For all subexpressions
18:                 yield s_0 P_1 s_1 P_2 ... P_k s_k   ▷ Substitute subexpressions into R's RHS
```

considers all available productions, and for each production, enumerates all combinations of arguments whose costs sum up to $\text{LVL} - 1$.

During search, each candidate expression is evaluated to see if it satisfies the examples (lines 5–7). Importantly, the search maintains a cache of all evaluation results E, and discard the newly constructed program if it is *observationally equivalent* to a program already in the bank (line 8), *i.e.* if it evaluates to the same output for all inputs in the examples. This step is the key to the efficiency of the bottom-up search algorithm: it allows the synthesizer to factorize the search space by evaluation result, significantly reducing the number of programs explored at each cost level.

6.3 The HYSYNTH Approach

A key challenge in program synthesis is the astronomical size of the search space the synthesizer has to explore. For example, to find the program Eq. 6.1, the solution to the ARC

task from the introduction, bottom-up search with a uniform weight function has to enumerate around 450K programs (all programs of size ≤ 16), which takes 4.5 minutes in our experiments.

On the other hand, sampling solutions to this task from an LLM yields programs that are *close* to the desired solution, even if not quite correct. As we show in Sec. 6.7, GPT-4o uses relevant components `update_color`, `color_of`, and `is_neighbor` in nearly all of its solutions (usually missing some part of the filter or using the wrong color in the transform), and never uses irrelevant components like `move` or `rotate`. This suggests that the LLM generally has the right intuition about the components the solution needs to use; our insight is to leverage this intuition to guide bottom-up search by *assigning lower weights to the components that the LLM uses frequently*.

6.3.1 Guiding Bottom-up Search with Context-Free LLM Approximation

The overview of our approach, HYSYNTH, is shown in Fig. 6.2. Given a PBE problem consisting of a DSL with grammar $\mathscr{G}$ and a set of input-output examples $\mathscr{E}$, HYSYNTH proceeds in three steps.

Step 1: Sampling Solutions from an LLM. HYSYNTH starts by creating an LLM prompt that contains $\mathscr{G}$ and $\mathscr{E}$; the prompt can be optionally augmented with in-context examples if they are available for the given DSL. A complete prompt for the ARC running example can be found in Sec. 6.8. The LLM is then used to sample a set $\{S_i\}_{i=1}^{N}$ of completions; the choice of N trades off computational cost and the faithfulness of the approximation to the true LLM conditional.

Step 2: Learning a PCFG from LLM Solutions. Next, HYSYNTH attempts to parse each completion S_i into a program P_i using the grammar $\mathscr{G}$. The resulting set of programs $\{P_i\}_{i=1}^{N'}$ (where $N' \leq N$) is used to learn a PCFG $\mathscr{G}_p$ via maximum likelihood estimation: $p(\mathrm{R}) = \frac{\mathrm{count}(\mathrm{R})+\alpha}{\sum_{\mathrm{R}\in\mathscr{R}}\mathrm{count}(\mathrm{R})+\alpha\times|\mathscr{R}|}$. Here $\mathrm{count}(\mathrm{R})$ is the frequency of rule R is all the derivations of the programs in $\{P_i\}$ and α is a smoothing parameter that ensures that every rule has a non-zero probability (typically set to 1).

Our experiments show that some models struggle to generate grammatical completions, leading to $N' \ll N$. To increase the sampling efficiency in those cases, HYSYNTH implements *non-strict mode*, where ungrammatical completions S_i are not discarded. Instead the tool performs lexical analysis on S_i to convert it into a sequence of terminals and approximates the frequency of each production R based on the frequency of its *operator terminal*, a designated terminal of R, which represents a DSL operator; *e.g.* count($Atom \rightarrow$ `not` $Atom$) = count(`not`).[3]

Step 3: Guiding Bottom-up Search with PCFG. Finally, HYSYNTH uses the PCFG computed in the previous step to derive a weighted grammar $\mathscr{G}_w$ as explained in Sec. 6.2.2, and uses it to initialize the bottom-up search procedure in Alg. 5. As a result, the search is guided by the insights from the the LLM. For example, the WCFG learned from the GPT-4o completions for the ARC task above gives the relevant transform operator `update_color` weight 2, while all other $Transform$ rules have weight 4; the relevant filter operators `color_of` and `is_neighbor` are similarly down-weighted. As a result, the search procedure only has to enumerate around 220K programs instead of 450K, achieving a 4x speedup, and solving the motivating example in just one minute with LLM guidance.

6.3.2 Domain-Specific Instantiations

We now describe how the HYSYNTH approach is instantiated in three different domains: ARC grid puzzles, TENSOR manipulations, and STRING manipulations.

ARC *Domain.* An example task from this domain is shown in Fig. 6.1a and has been used as a running example throughout this paper. There is no established DSL for ARC, and arguably, DSL design is the biggest challenge when attempting to solve ARC using a PBE approach, since it is hard to capture the wide variety of tasks in this domain. Our DSL is inspired by the rule-based language of ARGA [179], which we modified slightly to make it more compositional.

A program in our DSL is a sequence of rules of the form `if` *filter* `then` *transform*. A

[3]Typically, the operator terminal uniquely identifies R, but when this is not the case, we can normalize count(R) by the number of rules in $\mathscr{R}$ that produce this terminal.

rule refers to the current object `self`, which is modified by the transform if the filter is satisfied in the current state of the grid. The rule can also refer to other objects in the grid, such as `other` in Eq. 6.1. This program is well-defined because its filter uniquely identifies the object `other`; if the filter is too weak to uniquely determine the effect of the transform, the program's output is considered undefined. The full grammar of our DSL can be found in Sec. 6.14.

Instead of searching for a complete program using Alg. 5, we further optimize our synthesizer using a divide-and-conquer strategy inspired by [13], searching for filters and transforms *separately*. Specifically, HYSYNTH-ARC first searches for transforms that are correct on some objects in the grid; once it has found a set of transforms that collectively describe all grid objects, it searches for filters that distinguish between the subsets of objects changed by each transform.

Consider once again our running example. When the transform synthesizer enumerates the expression `update_color(color_of(other))`, it detects that this transform works for all *grey object*, because for each grey object `self` there exists a corresponding object `other` whose color can be copied. Now the goal of filter synthesis is to find a boolean expression that holds exactly for those pairs of objects (`self`, `other`) that make the transform work.

TENSOR *Domain.* This domain originates from the TFCODER synthesizer [144], which takes as input examples of a tensor transformation (with an optional natural language description) and synthesizes a TensorFlow program that performs the transformation. An example task is shown in Fig. 6.1b: `tf.gather_nd(in1, tf.stack((in2, in3), axis=-1))`. The main challenge, however, is that the TensorFlow grammar is very large (see Sec. 6.15), and most importantly, the programs are allowed to use an *unbounded* set of constants. The original TFCODER synthesizer requires the user to provide any non-standard constants that a task might require, and, according to their paper, this is the main barrier to the usability of their tool.

For program synthesis in this domain we use the TFCODER synthesizer off the shelf. TFCODER performs weighted bottom-up search, using a combination of hand-tuned weights

and weights derived by two custom-trained neural models. HYSYNTH-TENSOR replaces these weights entirely with weights computed by sampling from an LLM. Importantly, our version of the tool does not require the user to provide any constants; instead we extract constants from the LLM completions, whereby significantly reducing the burden on the user.

STRING *Domain.* Our third domain involves string manipulation tasks from the SYGUS competition [10], which are inspired by spreadsheet use cases. An example task, which requires extracting the top-level domain name from a URL, is shown in Fig. 6.1c. In this domain we use the PROBE [21] synthesizer off the shelf. PROBE performs weighted bottom-up search, starting with a uniform grammar and updating the weights on the fly; HYSYNTH-STRING instead initializes PROBE's search with weights derived from an LLM, and disables the weight updates during search.

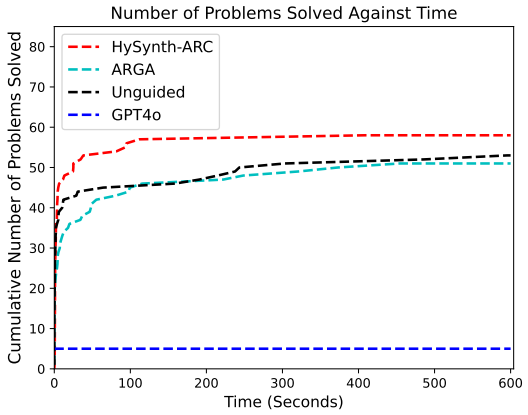
6.4 Experiments and Results

6.4.1 Experimental Setup

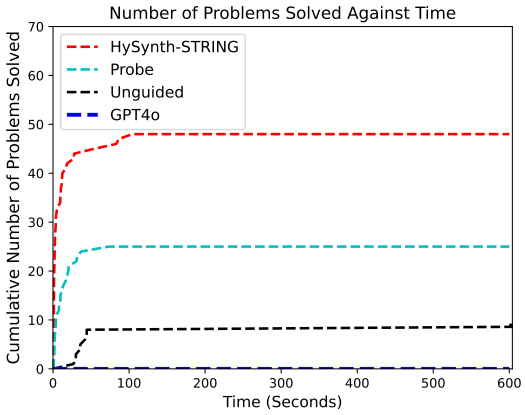
We evaluate HYSYNTH on 299 PBE tasks from three different domains: ARC (160 tasks), STRING (70 tasks) and TENSOR (69 tasks).

ARC *Benchmark.* The 160 ARC tasks are taken from the testing set of ARGA [179]. This *object-centric* subset of the full ARC corpus is known as OBJECT-ARC, and has been used to evaluate other ARC solvers [104]. ARC specifications consist of 2-7 input-output training grids and 1 testing grid. Correctness is based on whether the generated solution produces the correct output on the testing grid. Our ARC DSL has a total of 20 operations and 50 constants and variables across all types.

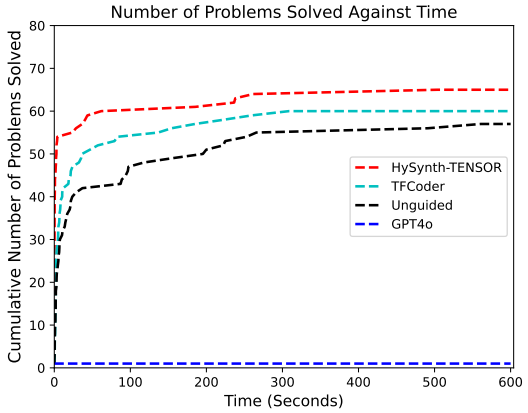
TENSOR *Benchmark.* The 69 TENSOR tasks taken from TFCODER focus on tensor manipulation. 49 of them are sourced from StackOverflow inquiries, and 20 are from real-world scenarios faced by TensorFlow users at Google. The overall benchmark suite consists of 72 tasks. We use three of these tasks as in-context examples and evaluate on the rest. The grammar for



(a) HYSYNTH-ARC results with GPT4O



(b) HYSYNTH-STRING results with GPT4O



(c) HYSYNTH-TENSOR results with GPT4O

Domain/Model	% Valid completions
TENSOR-GPT4O	99.96%
TENSOR-DEEPSEEK	92.8%
STRING-GPT4O	98.3%
STRING-DEEPSEEK	86%
ARC-GPT4O	78%

(d) Percentage of syntactically valid completions.

Figure 6.4: (a,b,c) Number of benchmarks solved by HYSYNTH as a function of time for the ARC, TENSOR, and STRING domains; timeout is 10 min. (d) Percentage of syntactically valid completions per domain.

this domain consists of 134 Tensorflow operations, primitives like `0, 1, -1, True` and other task-specific constants.

STRING *Benchmark.* The 70 STRING tasks are taken from testing set of PROBE, which is derived them from the SYGUS benchmark [10]. The number of examples ranges from 2 to 400. The original SYGUS benchmark have custom grammars for each task, but we use a union of all the grammars to make the search more challenging; the union grammar has 16 operations and 59 constants.

Configurations. Our main HYSYNTH configuration uses GPT4O as the LLM, with 100

samples per task to learn a PCFG in non-strict mode (*i.e.* syntactically invalid completions are included in the PCFG learning process, as explained in Sec. 6.3.1). For each domain, we compare the performance of HYSYNTH with a baseline synthesizer for that domain (ARGA[4], PROBE, and TFCODER), as well as two ablations:

(1) *no search*, *i.e.* using the 100 samples from the LLM directly, and

(2) *unguided search*, *i.e.* running the same synthesizer but with a uniform weighted grammar.

We also analyze the performance of HYSYNTH with different numbers of samples used to learn the PCFG (10, 20, and 50), with other LLMs (GPT3.5 and DEEPSEEK [79]), as well as in strict mode (which discards syntactically invalid LLM completions). Search timeout is set to 10 minutes for all experiments.

6.4.2 Results

How does* HYSYNTH *compare to baselines and ablations?. We compare the time to solution for the main HYSYNTH configuration, baseline synthesizers, and the two ablations; the results for the three domains are shown in Fig. 6.4a, Fig. 6.4c, and Fig. 6.4b. Overall, HYSYNTH consistently outperforms both the baseline synthesizers and ablations, solving more tasks across all domains and time scales.

In more detail, direct LLM sampling performs very poorly on all domains, solving between 0 and 5 tasks; this confirms our hypothesis that LLMs struggle on PBE tasks in domain-specific languages, which are not prevalent in their training data. Interestingly, despite not being able to solve *any* STRING tasks by itself, GPT4O provides excellent guidance for HYSYNTH on that domain, helping it solve 5x more tasks than the unguided search!

In STRING and TENSOR domains, the baseline synthesizers predictably do better than unguided search, since both use the same search implementation, but with different weights. On

[4]At the time of writing, ARGA is no longer state of the art on the OBJECT-ARC dataset; we explain in Sec. 6.5 why the comparison with ARGA is still relevant.

ARC, however, our custom synthesizer outperforms ARGA[5] even without LLM guidance; this speaks to the efficiency of the bottom-up search and the divide-and-conquer strategy we use, which are results of years of research in the program synthesis community.

How many samples are needed to learn a PCFG?. To better understand how the number of samples affects the quality of PCFG guidance, we vary the number of GPT4O programs used in PCFG learning $N = 10, 20, 50, 100$, and once again measure the number of tasks solved over time. The results are shown in Fig. 6.8 in Sec. 6.9. As expected, larger sample sizes generally lead to better performance, but the difference is minimal: in ARC and TENSOR, the difference between the best and worst performing versions of HYSYNTH is only 2 and 1 problems, respectively, while in STRING, HYSYNTH solves 5 fewer problems with 10 samples than with 100. Despite these differences, all versions of HYSYNTH still outperform the baseline and unguided search. This suggests that fewer samples are sufficient to effectively train a robust surrogate model, thereby optimizing costs.

Do our results generalize to other models?. To answer this question, we repeat our experiments on STRING and TENSOR domains with GPT3.5 and the open-source model `deepseek-coder-33b-instruct` (DEEPSEEK) [79]. The results with these models are detailed in Fig. 6.9 in Sec. 6.10, and they corroborate the pattern observed with GPT4O, where the guided versions outperform the baseline, unguided search, and direct sampling from the LLM.

How important is non-strict mode?. Fig. 6.4d shows the percentage of syntactically valid completions generated by GPT4O and DEEPSEEK (where applicable). You can see that while on TENSOR almost all completions are valid, this percentage falls to 78% for ARC and 86% for STRING; this is not surprising, given that the former are TensorFlow programs, which the model has seen during training, while the latter two are custom DSLs. Hence our non-strict mode proves especially helpful for low-resource domains, where otherwise we would have to

[5] [179] report 57 tasks for ARGA but we could only reproduce 51 on our hardware with a 10 minute timeout.

discard a large proportion of completions. At the same time, we find that *given the same number of completions to learn from*, the PCFGs learned in non-strict mode are just as effective as those learned in strict mode: for example, HYSYNTH-TENSOR with the guidance from 100 DEEPSEEK completions solves 67 tasks *in either mode* (with the difference that strict mode has to sample more completions to get 100 valid ones).

6.4.3 Limitations

The main limitation of our hybrid approach *wrt.* to purely neural approaches is that is requires implementing a synthesizer for each DSL of interest; although we have shown that the same bottom-up search can be used across different domains, some implementation effort is still required. On the other hand, compared to purely symbolic approaches, our method requires sampling from an LLM, which is costly; additionally, the guidance provided by our approach is only as good as the LLM's completions: if they contain many irrelevant operators, our guided search can be *slower* than unguided search. Finally, our experiments are subject to the usual threat that the LLMs might have seen our benchmarks in their training data; we do not consider it a major issue, however, given that our main result is the superior performance of guided search *relative* to using LLMs without search.

6.5 Related Work

Guiding Program Synthesis with Probabilistic Models. The traditional approach to *program synthesis* is based on combinatorial search [15], augmented with pruning techniques based on program semantics [159, 5, 13]. To further speed up the search, researchers have proposed *guiding* the search with a learned probabilistic model. Most approaches to guided search use special-purpose models that have to be trained on a domain-specific corpus of programs [102] or PBE tasks [19, 94, 122, 145]. Although some of these models can be trained on synthetic data, the training process is still expensive and requires manual tuning, which makes it hard to apply these techniques to new domains.

With the advent of pretrained Large Language Models (LLMs), it seems only natural to use them to guide search-based program synthesis, thus alleviating the need for domain-specific training data. We are only aware of one other attempt to do this: concurrent work by [107], which also extracts a PCFG from the LLM's samples, similarly to PROBE. An important difference is that they use the PCFG to guide *top-down* A* search, while we use it to guide *bottom-up* search, which is known to be more efficient (they also evaluate their tool on synthesis from logical formulas as opposed to PBE).

Solving the Abstraction and Reasoning Corpus. All state-of-the-art solvers for this benchmark have relied on carefully curated DSLs for ARC [27, 176, 6, 104, 61]. [179] proposed the DSL we extend in our approach, and the OBJECT-ARC subset we evaluate on. [104] embed their DSL as a subset of PDDL and use a Generalized Planning (GP) algorithm as their search component. They have the current best performance on OBJECT-ARC, however they encode more domain-knowledge in the form of preconditions and per-abstraction restrictions on filters and transforms, to make GP viable. Our approach does not require this additional information. [6, 20] use DreamCoder [53], to perform execution-guided search over a DSL for grid manipulations, however they only provide proof-of-concept evaluations. [167, 156] also use an LLM to generate code given the spec of the task. Both of these approaches interact with the model across several rounds, while our technique uses the suggestions from the LLM only as a starting point. Our technique also performs a complete search guided by the LLM distribution, enabled by the structure of our DSL, whereas previous approaches only consider code directly generated by the LLM.

6.6 Conclusion and Future Work

Our approach introduces a robust technique for using both valid and invalid completions from an LLM to learn a surrogate model. By incorporating ungrammatical completions, we can extract useful insights that would otherwise be discarded. Overall, we provide an alternative to

```
// Solution 1, occurs 6 times
if color_of(self) = GREY ∧ is_neighbor(self, other)
  then update_color(color_of(other))

// Solution 2, occurs 1 time
if is_neighbor(self, other) ∧ color_of(other) = GREY
  then update_color(color_of(other))

// Solution 3, occurs 1 time
if color_of(self) = GREY
  then update_color(color_of(other))

// Solution 4, occurs 1 time
if not (color_of(self) = GREY) ∧ is_neighbor(self, other) ∧ color_of(
    other) = GREY
  then update_color(FUCHSIA)

// Solution 5, occurs 1 time
if size_of(self) = 4 then update_color(RED) ;
if size_of(self) = 4 ∧ color_of(self) = GREY then update_color(FUCHSIA)
    ;
if size_of(self) = 4 ∧ color_of(self) = BLUE then update_color(ORANGE) ;
if size_of(self) = 4 ∧ color_of(self) = YELLOW then update_color(CYAN)
```

Figure 6.5: Ten samples from GPT4o for the motivating example in Fig. 6.1a

the conventional strategy of large-scale sampling from LLMs, proposing a more effective use of the available completions to guide the search process. An interesting future direction would be to guide search with a more expressive context-dependent surrogate model.

6.7 GPT4o Solutions for the Motivating Example

Recall the motivating example in Fig. 6.1a where the task is to update the color of the grey objects to the color of their single-pixel neighbor. As a reminder, the smallest correct solution to this task consists of the following rule:

```
if color_of(self) = GREY ∧ is_direct_neighbor(self, x) ∧ size_of(x) =
    MIN
  then update_color(color_of(x))
```

Fig. 6.5 shows the programs we obtained by deduplicating 10 samples from GPT4o for this task. The syntax of the solutions is slightly modified for readability; our implementation uses a

```
You are an assistant chatbot with human-like perception, reasoning and
   learning capabilities.
You can solve tasks concisely, efficiently, and moreover, correctly.
Let's engage in perception- and logic-based tasks.
You only output source code.
No explanations or any other text.
Only code.
```

Figure 6.6: System prompt for ARC domain.

LISP-style s-expression syntax [114] to simplify parsing.

As you can see, the most frequent solution is almost correct, except that is does not constrain the neighbor `other` to be of size 1; this leads to the constraint being ambiguous (since every gray object has multiple neighbors of different colors), in which case the program semantics is considered undefined. That said, you can observe that the model consistently uses relevant components, such as `color_of`, `is_neighbor`, and `update_color`, which enables us to extract a useful PCFG from these solutions.

When we increased the sample size to 125, GPT4o was able to produce one correct solution (which is slightly larger than the minimal solution above):

```
if color_of(self) = GREY ∧ is_neighbor(self, other) ∧ not (color_of(
   other) = GREY)
 then update_color(color_of(other))
```

6.8 LLM Prompt for the Motivating Example

6.8.1 System Prompt

The system prompt given to the LLM for ARC domain is shown in Fig. 6.6.

6.8.2 User Prompt

The full user prompt for the ARC domain is shown in Fig. 6.7. It contains the domain-specific language, four in-context examples and the query for the test task.

```
You are an efficient assistant for logical reasoning and code generation .
You will help me solve a visual perception and reasoning task.
I will first provide you with the definition of a Domain Specific Language you will use for writing a solution for the task.
I will then present you with the description of the task that you will be tested in.
You will then respond the queries I make regarding the solution of the task.

This is the definition of the DSL you will use to solve the task.
It is given as a context-free grammar in the EBNF format used by the Lark parser generator, with some informative comments about the semantics.
You will return a string that is parseable by the ‘program‘ non-terminal of the grammar.

‘‘‘
library: "(" program* ")"

// Rules are executed one after another, in the order they appear.
// There could be no rules, in which case the program does nothing.
program: "(" "do" rule* ")"
...

<<< DSL IMPLEMENTATION IN LARK >>>

Now we continue with the visual perception and reasoning task.
The input for the task is a small number of pairs of grids of characters .
The value of each of the cells of the grids are the colors defined in the DSL, so we can think of grids as images.
Each pair of images correspond to an input-output example for an unknown program P.
For each pair, the program P is evaluated on the image grid and operates on the objects that appear in it.
The output of the program is then the output image.
The objects in the images are easy and natural to identify for humans, so there is no need to define them explicitly.
However you are able to abstract them correctly, and the DSL is interpreted with the same correct abstraction.

Now I will show you some demonstration tasks along with the output you would be expected to produce for each of them.

## DEMONSTRATION TASK 1

### INPUT
PAIR 1
INPUT GRID:
0 0 0 0 0 R 0 0
0 0 0 0 0 R 0 0
```

Figure 6.7: User prompt for ARC domain.

```
OUTPUT GRID:
0 0 0 0 0 Y 0 0
0 0 0 0 0 Y 0 0

<<< ENCODING OF EXAMPLE PAIR 2 AND 3 OF DEMO TASK 1>>>

### EXPECTED OUTPUT
{
    "nl_description": "Recolor all objects to color Y",
    "code": <<< EXPECTED CODE IN DSL >>>
}

<<< MORE DEMONSTRATION TASKS (4 IN TOTAL) >>>

Now follows task you will be evaluated on.
Output the solution as a JSON object, which should contain both a
    ↪ natural language description of the solution and the solution
    ↪ written in the DSL.
The code should be parseable by the DSL grammar.
The JSON must have the following structure:

{
    "nl_description": "TO_BE_FILLED",
    "code": "TO_BE_FILLED"
}

## TEST TASK

PAIR 1
INPUT GRID:
0 0 R 0 0 F 0 0 0 C
0 0 0 0 0 0 0 0 0 0
0 0 0 0 X X X X 0 0
0 0 0 0 X X X X 0 0
0 X X 0 X X X X 0 0
0 X X 0 X X X X 0 0
0 X X 0 0 0 0 0 0 0
0 X X 0 0 0 0 X X X
0 X X 0 0 0 0 X X X
0 0 0 0 0 0 0 X X X
OUTPUT GRID:
0 0 R 0 0 F 0 0 0 C
0 0 0 0 0 0 0 0 0 0
0 0 0 0 F F F F 0 0
0 0 0 0 F F F F 0 0
0 R R 0 F F F F 0 0
0 R R 0 F F F F 0 0
0 R R 0 0 0 0 0 0 0
0 R R 0 0 0 0 C C C
0 R R 0 0 0 0 C C C
0 0 0 0 0 0 0 C C C

<<< REST OF THE I/O EXAMPLES OF TEST TASK >>>
```

6.9 Different sample sizes ablation for ARC, TENSOR and STRING domains

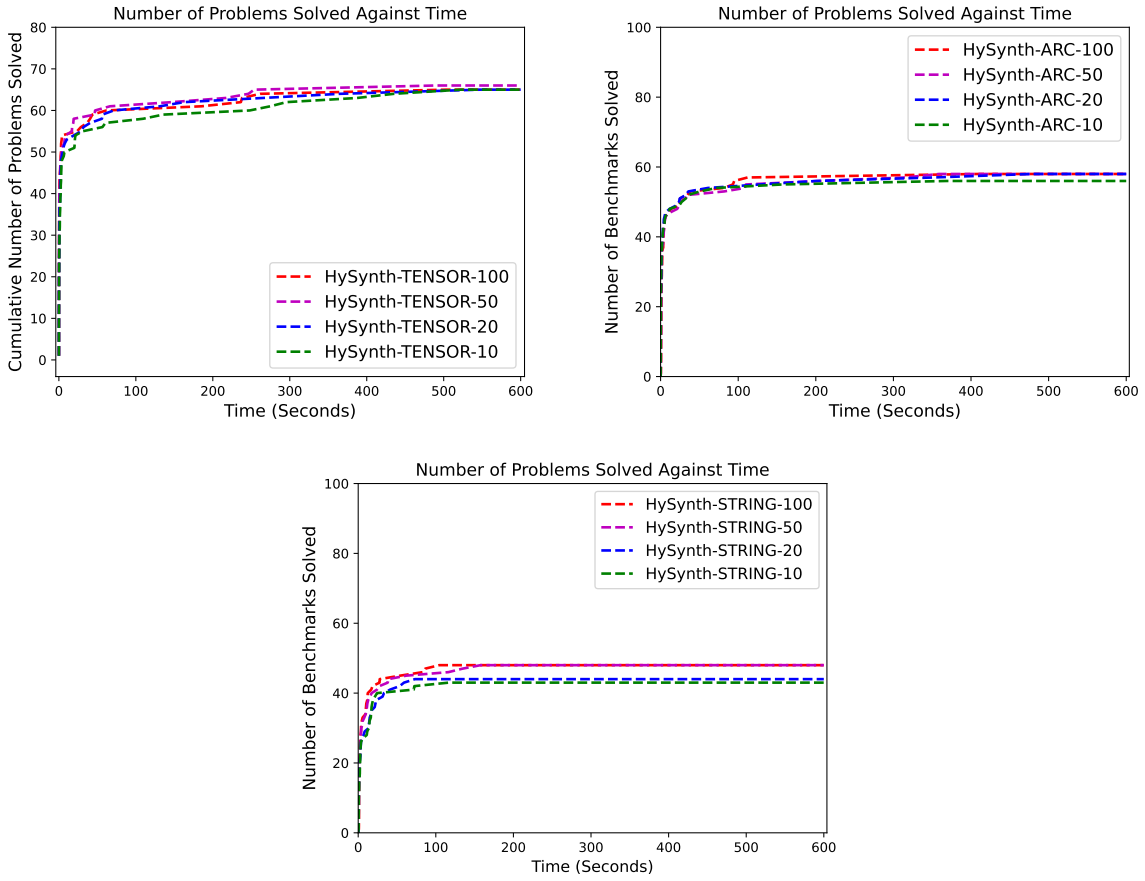
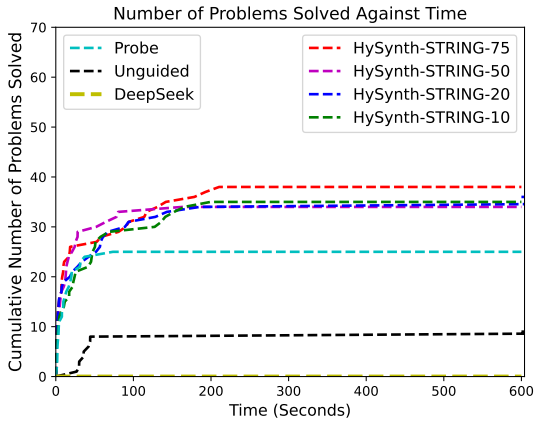
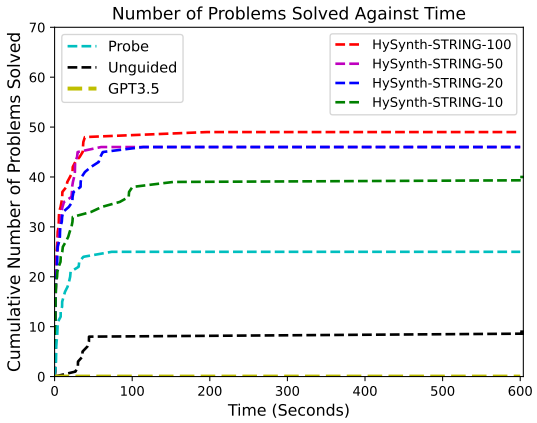


Figure 6.8: HYSYNTH-ARC, HYSYNTH-TENSOR and HYSYNTH-STRING results guided by a PCFG learned from different number of GPT4O samples (n=10, 20, 50, 100).

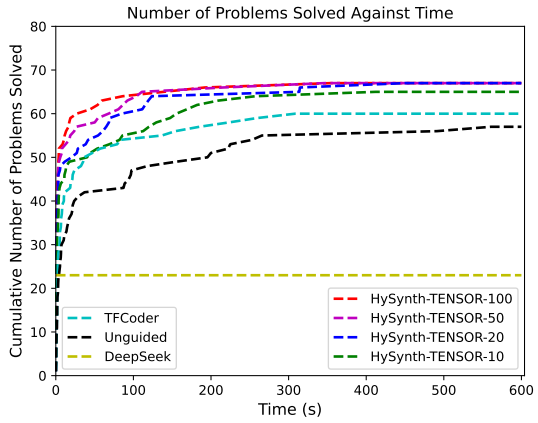
6.10 Experimental results with LLMs DEEPSEEK and GPT3.5



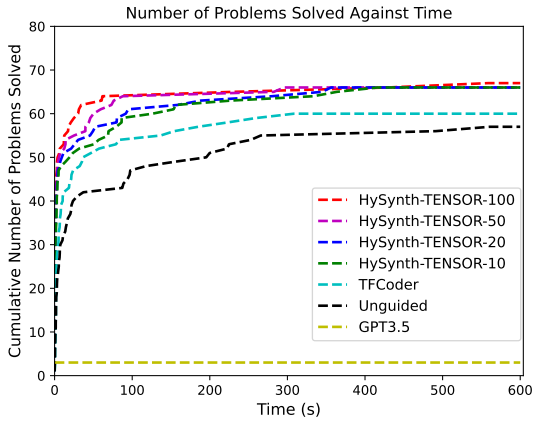
(a) HYSYNTH-STRING results with DEEPSEEK



(b) HYSYNTH-STRING results with GPT3.5



(c) HYSYNTH-TENSOR results with DEEPSEEK



(d) HYSYNTH-TENSOR results with GPT3.5

Figure 6.9: HYSYNTH-STRING and HYSYNTH-TENSOR results with DEEPSEEK and GPT3.5.

```
You are a coding assistant. Be precise and terse.
You will be provided a list of tensorflow operators, a task description,
    and some input/output examples.
Your task is to generate the body of a python function that will
   transform the input to the output.
Only use the operators provided in the list.
Your answer should be as short as possible while still being correct.
Make sure to only generate python code.
```

Figure 6.10: System prompt for TENSOR domain.

6.11 LLM Prompt for the TENSOR Grammar

The system and user prompt for TENSOR domain are in Fig. 6.10 and Fig. 6.11.

```
[TENSORFLOW OPERATORS]
<<< see appendix E >>>

[TASK DESCRIPTION]
index into the tensor

[INPUTS]
[[ 5.  2.]
 [ 1.  3.]
 [ 0. -1.]]


[OUTPUTS]
[[[ 5.  5.]
  [ 1.  1.]
  [ 0.  0.]]

 [[ 2.  2.]
  [ 3.  3.]
  [-1. -1.]]]

[PROGRAM]
def transform(in1):
```

Figure 6.11: User prompt for TENSOR domain

6.12 LLM Prompt for STRING

The system and user prompt for STRING domain are in Fig. 6.12 and Fig. 6.13.

6.13 The Full STRING Grammar

The full grammar of our STRING DSL is shown in Sec. 6.13.

```
You are a coding assistant. Be precise and terse.
You will be given a SyGuS grammar, a natural language specification, and
    ↪ a set of input-output examples.
Your task is to complete the provided function definition with an
    ↪ implementation that is correct according to the grammar,
    ↪ specification, and examples.
Your answer should be as short as possible while still being correct.
Make sure that your answer is a valid s-expression.
```

Figure 6.12: System prompt for STRING domain

```
[GRAMMAR]
(synth-fun f ((_arg_0 String)) String ((Start String (ntString)) (
   ↪ ntString String (_arg_0 "" " " "BRD" "DRS" "LDS" "Branding" "
   ↪ Direct Response" "Leads" "=" "/" "in" "_" "9" "." "microsoft" "
   ↪ windows" "apple" "mac" "-" "1" "2" "3" "4" "5" "6" "7" "8" "0" ","
   ↪  "<" ">" "/n" "%" "b" "apple" "bananas" "strawberries" "oranges" "
   ↪ LLC" "Inc" "Corporation" "Enterprises" "Company" "(" ")" "+" "name
   ↪ " "," (str.++ ntString ntString) (str.replace ntString ntString
   ↪ ntString) (str.at ntString ntInt) (int.to.str ntInt) (ite ntBool
   ↪ ntString ntString) (str.substr ntString ntInt ntInt))) (ntInt Int
   ↪ (-1 1 2 3 4 5 6 7 8 9 0 1 0 -1 (+ ntInt ntInt) (- ntInt ntInt) (
   ↪ str.len ntString) (str.to.int ntString) (ite ntBool ntInt ntInt) (
   ↪ str.indexof ntString ntString ntInt))) (ntBool Bool (true false (=
   ↪  ntInt ntInt) (str.prefixof ntString ntString) (str.suffixof
   ↪ ntString ntString) (str.contains ntString ntString)))))

[NATURAL LANGUAGE SPECIFICATION]
; https://exceljet.net/formula/get-top-level-domain-tld

[EXAMPLES]
www.domain.com → com
mail.net → net
www.amazon.co.uk → uk

[SOLUTION]
(define-fun f (_arg_0 String) String
```

Figure 6.13: User message for STRING

Start →	*S*	
S →	`arg0` | `arg1` | ...	string variables
|	`lit-1` | `lit-2` | ...	string literals
|	(`replace` *S S S*)	`replace s x y` replaces first occurrence of `x` in `s` with `y`
|	(`concat` *S S*)	`concat x y` concatenates `x` and `y`
|	(`substr` *S I I*)	`substr x y z` extracts substring of length `z`, from index `y`
|	(`ite` *B S S*)	`ite x y z` returns `y` if `x` is true, otherwise `z`
|	(`int.to.str` *I*)	`int.to.str x` converts int `x` to a string
|	(`at` *S I*)	`at x y` returns the character at index `y` in string `x`
B →	`true` | `false`	bool literals
|	(`=` *I I*)	`= x y` returns true if `x` equals `y`
|	(`contains` *S S*)	`contains x y` returns true if `x` contains `y`
|	(`suffixof` *S S*)	`suffixof x y` returns true if `x` is the suffix of `y`
|	(`prefixof` *S S*)	`prefixof x y` returns true if `x` is the prefix of `y`
I →	`arg0` | `arg1` | ...	int variables
|	`lit-1` | `lit-2` | ...	int literals
|	(`str.to.int` *S*)	`str.to.int x` converts string `x` to a int
|	(`+` *I I*)	`+ x y` sums `x` and `y`
|	(`-` *I I*)	`- x y` subtracts `y` from `x`
|	(`length` *S*)	`length x` returns length of `x`
|	(`ite` *B I I*)	`ite x y z` returns `y` if `x` is true, otherwise `z`
|	(`indexof` *S S I*)	`indexof x y z` returns index of `y` in `x`, starting at index `z`

Figure 6.14: The full SyGuS String grammar of the Probe benchmark suite. Integer and string variables and constants change per benchmark. Some benchmark files contain a reduced grammar.

6.14 The Full ARC DSL

The full grammar of our ARC DSL is shown in Fig. 6.15 (for filters) and Fig. 6.16 (for transforms).

$$
\begin{array}{rll}
\mathit{Start} \rightarrow & \mathit{Filters} & \\
\mathit{Filters} \rightarrow & \mathit{Filter\_Ops} & \\
\mid & \texttt{And}\ \mathit{Filters}\ \mathit{Filters} & \\
\mid & \texttt{Or}\ \mathit{Filters}\ \mathit{Filters} & \\
\mid & \texttt{Not}\ \mathit{Filters} & \\
\mathit{Filter\_Ops} \rightarrow & \mathit{Color} == \mathit{Color} & \\
\mid & \mathit{Size} == \mathit{Size} & \\
\mid & \mathit{Degree} == \mathit{Degree} & \\
\mid & \mathit{Height} == \mathit{Height} & \\
\mid & \mathit{Width} == \mathit{Width} & \\
\mid & \mathit{Row} == \mathit{Row} & \\
\mid & \mathit{Column} == \mathit{Column} & \\
\mid & \mathit{Shape} == \mathit{Shape} & \\
\mid & \mathit{Obj} == \mathit{Obj} & \\
\mathit{Color} \rightarrow & \texttt{Black} \mid \texttt{Blue} \mid \texttt{Yellow} \mid \texttt{Red} \mid \texttt{Green} \mid \texttt{Grey} \mid \texttt{Fuchsia} \mid & \\
\mid & \texttt{Orange} \mid \texttt{Cyan} \mid \texttt{Brown} \mid \texttt{Color\_Of(Obj)} & \text{object colors} \\
\mathit{Size} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Size\_Of(Obj)} & \text{object sizes} \\
\mathit{Degree} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Degree\_Of(Obj)} & \text{graph degrees} \\
\mathit{Height} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Height\_Of(Obj)} \mid \ldots & \text{object heights} \\
\mathit{Width} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Width\_Of(Obj)} \mid \ldots & \text{object widths} \\
\mathit{Column} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Center} \mid \texttt{Column\_Of(Obj)} \mid \ldots & \text{grid columns} \\
\mathit{Row} \rightarrow & \texttt{Max} \mid \texttt{Min} \mid \texttt{Odd} \mid \texttt{Center} \mid \texttt{Row\_Of(Obj)} \mid \ldots & \text{grid rows} \\
\mathit{Shape} \rightarrow & \texttt{Enclosed} \mid \texttt{Square} \mid \texttt{Shape\_Of(Obj)} \mid \ldots & \text{object shapes} \\
\mathit{Obj} \rightarrow & \texttt{obj-0} \mid \texttt{obj-1} \mid \texttt{obj-2} \mid \texttt{Neighbor\_Of(Obj)} \mid \ldots &
\end{array}
$$

Figure 6.15: The modified filter grammar derived from ARGA [179], object specific parameters like size, degree, height, width change per benchmark.

$$
\begin{aligned}
\mathit{Start} &\rightarrow \mathit{Transforms} \\
\mathit{Transforms} &\rightarrow \mathit{Transform\_Ops} \\
&\mid \mathit{Transform\_Ops}\ \mathit{Transforms} \\
\mathit{Transform\_Ops} &\rightarrow \texttt{Update\_Color}\ \mathit{Color} \\
&\mid \texttt{Move\_Node}\ \mathit{Direction} \\
&\mid \texttt{Move\_Node\_Max}\ \mathit{Direction} \\
&\mid \texttt{Extend\_Node}\ \mathit{Direction}, \mathit{Overlap} \\
&\mid \texttt{Rotate\_Node}\ \mathit{Angle} \\
&\mid \texttt{Add\_Border}\ \mathit{Color} \\
&\mid \texttt{Fill\_Rectangle}\ \mathit{Color}, \mathit{Overlap} \\
&\mid \texttt{Hollow\_Rectangle}\ \mathit{Color} \\
&\mid \texttt{Mirror}\ \mathit{Axis} \\
&\mid \texttt{Flip\_Node}\ \mathit{Axis} \\
&\mid \texttt{NoOp} \\
\mathit{Color} &\rightarrow \texttt{Black} \mid \texttt{Blue} \mid .. \mid \texttt{Green} \mid \texttt{Grey} \mid \texttt{Fuchsia} \mid \texttt{Orange} \mid \texttt{Color\_of(Object)} \\
\mathit{Direction} &\rightarrow \texttt{Left} \mid \texttt{Right} \mid \texttt{Up} \mid \texttt{Down} \mid .. \mid \texttt{DownLeft} \mid \texttt{DownRight} \mid \texttt{Dir\_of(Object)} \\
\mathit{Overlap} &\rightarrow \texttt{True} \mid \texttt{False} \mid \\
\mathit{Angle} &\rightarrow \texttt{90} \mid \texttt{180} \mid \texttt{270} \\
\mathit{Axis} &\rightarrow \texttt{Vertical} \mid \texttt{Horizontal} \mid \texttt{LeftDiagonal} \mid \texttt{RightDiagonal} \mid \\
\mathit{Object} &\rightarrow \texttt{obj-0} \mid \texttt{obj-1} \mid \texttt{obj-2} \mid ...
\end{aligned}
$$

Figure 6.16: The modified transform grammar derived from ARGA [179], parameters like objects change based on the benchmark.

6.15 The Full TENSOR Grammar

```
General TensorFlow functions:
-----------------------------
tf.abs(x)
tf.add(x, y)
tf.add_n(inputs)
tf.argmax(input, axis)
tf.argmin(input, axis)
tf.argsort(values, axis, stable=True)
tf.argsort(values, axis, direction='DESCENDING', stable=True)
tf.boolean_mask(tensor, mask)
tf.broadcast_to(input, shape)
tf.cast(x, dtype)
tf.clip_by_value(t, clip_value_min, clip_value_max)
tf.concat(values, axis)
tf.constant(value)
tf.constant(value, dtype)
tf.divide(x, y)
tf.equal(x, y)
tf.exp(x)
tf.expand_dims(input, axis)
tf.eye(num_rows)
tf.eye(num_rows, num_columns)
tf.eye(num_rows, dtype)
tf.fill(dims, value)
tf.gather(params, indices)
```

Figure 6.17: List of TensorFlow operations as used in TFCODER.

```
tf.gather(params, indices, axis, batch_dims)
tf.gather_nd(params, indices)
tf.gather_nd(params, indices, batch_dims)
tf.greater(x, y)
tf.greater_equal(x, y)
tf.math.bincount(arr)
tf.math.ceil(x)
tf.math.count_nonzero(input)
tf.math.count_nonzero(input, axis)
tf.math.cumsum(x, axis)
tf.math.cumsum(x, axis, exclusive=True)
tf.math.divide_no_nan(x, y)
tf.math.floor(x)
tf.math.log(x)
tf.math.negative(x)
tf.math.reciprocal(x)
tf.math.reciprocal_no_nan(x)
tf.math.segment_max(data, segment_ids)
tf.math.segment_mean(data, segment_ids)
tf.math.segment_min(data, segment_ids)
tf.math.segment_prod(data, segment_ids)
tf.math.segment_sum(data, segment_ids)
tf.math.squared_difference(x, y)
tf.math.top_k(input, k)
tf.math.unsorted_segment_max(data, segment_ids, num_segments)
tf.math.unsorted_segment_mean(data, segment_ids, num_segments)
tf.math.unsorted_segment_min(data, segment_ids, num_segments)
tf.math.unsorted_segment_prod(data, segment_ids, num_segments)
tf.math.unsorted_segment_sum(data, segment_ids, num_segments)
tf.matmul(a, b)
tf.maximum(x, y)
tf.minimum(x, y)
```

```
tf.one_hot(indices, depth)
tf.ones(shape)
tf.ones_like(input)
tf.pad(tensor, paddings, mode='CONSTANT')
tf.pad(tensor, paddings, mode='CONSTANT', constant_values)
tf.pad(tensor, paddings, mode='REFLECT')
tf.pad(tensor, paddings, mode='SYMMETRIC')
tf.range(start)
tf.range(start, limit, delta)
tf.reduce_any(input_tensor, axis)
tf.reduce_max(input_tensor)
tf.reduce_max(input_tensor, axis)
tf.reduce_mean(input_tensor)
tf.reduce_mean(input_tensor, axis)
tf.reduce_min(input_tensor)
tf.reduce_min(input_tensor, axis)
tf.reduce_prod(input_tensor, axis)
tf.reduce_sum(input_tensor)
tf.reduce_sum(input_tensor, axis)
tf.reshape(tensor, shape)
tf.reverse(tensor, axis)
tf.roll(input, shift, axis)
tf.round(x)
tf.searchsorted(sorted_sequence, values, side='left')
tf.searchsorted(sorted_sequence, values, side='right')
tf.sequence_mask(lengths)
tf.sequence_mask(lengths, maxlen)
tf.shape(input)
tf.sign(x)
tf.sort(values, axis)
tf.sort(values, axis, direction='DESCENDING')
```

```
tf.squeeze(input)
tf.squeeze(input, axis)
tf.stack(values, axis)
tf.subtract(x, y)
tf.tensordot(a, b, axes)
tf.tile(input, multiples)
tf.transpose(a)
tf.transpose(a, perm)
tf.unique_with_counts(x)
tf.unstack(value, axis)
tf.where(condition)
tf.where(condition, x, y)
tf.zeros(shape)
tf.zeros_like(input)
SparseTensor functions:
-----------------------
tf.SparseTensor(indices, values, dense_shape)
tf.sparse.add(a, b)
tf.sparse.concat(axis, sp_inputs)
tf.sparse.expand_dims(sp_input, axis)
tf.sparse.from_dense(tensor)
tf.sparse.maximum(sp_a, sp_b)
tf.sparse.minimum(sp_a, sp_b)
tf.sparse.reduce_max(sp_input, axis, output_is_sparse)
tf.sparse.reduce_sum(sp_input, axis, output_is_sparse)
tf.sparse.reset_shape(sp_input)
tf.sparse.reshape(sp_input, shape)
tf.sparse.retain(sp_input, to_retain)
tf.sparse.slice(sp_input, start, size)
tf.sparse.split(sp_input, num_split, axis)
tf.sparse.to_dense(sp_input)
tf.sparse.to_dense(sp_input, default_value)
```

```
tf.sparse.to_indicator(sp_input, vocab_size)
tf.sparse.transpose(sp_input)
tf.sparse.transpose(sp_input, perm)

Python-syntax operations:
-------------------------
IndexingAxis1Operation: arg1[:, arg2]
IndexingOperation: arg1[arg2]
PairCreationOperation: (arg1, arg2)
SingletonTupleCreationOperation: (arg1,)
SlicingAxis0BothOperation: arg1[arg2:arg3]
SlicingAxis0LeftOperation: arg1[arg2:]
SlicingAxis0RightOperation: arg1[:arg2]
SlicingAxis1BothOperation: arg1[:, arg2:arg3]
SlicingAxis1LeftOperation: arg1[:, arg2:]
SlicingAxis1RightOperation: arg1[:, :arg2]
TripleCreationOperation: (arg1, arg2, arg3)
```

6.16 Detailed Prompt Settings

For ARC, we sample completions with temperature 1 and 4000 max tokens. For TENSOR, we use temperature 1 and 300 max tokens. For SYGUS, we use temperature 0.5 and 200 max tokens. We use the same settings for all 3 LLMs. When prompting GPT4O for ARC, we set `response_type` to JSON.

6.17 Broader Research Impacts

Our method presents a powerful strategy for harnessing both syntactically valid and invalid outputs from an LLM to learn a surrogate model. Incorporating hallucinatory outputs – often erroneous generated by the model, allows us to extract insights that are discarded in

standard practices. Our approach mitigates the need for large-scale sampling of completions from LLMs, promoting a more efficient and effective utilization of these models, saving resources. Our method not only improves the cost effectiveness of using LLMs but also opens up new avenues for enhancing model robustness and adaptability across different domains.

6.18 Acknowledgements

Chapter 6, in full, has been submitted for publication of the material as it may appear in HySynth: Context-Free LLM Approximation for Guiding Program Synthesis. Barke, Shraddha; Gonzalez, Emmanuel; Kasibatla, Saketh; Berg-Kirkpatrick, Taylor; and Polikarpova, Nadia. The dissertation author was the primary investigator and author of this paper.

Bibliography

[1] Kaggle datasets.

[2] Maximum coverage problem.

[3] Euphony benchmark suite, 2018.

[4] Arc kaggle competition leaderboard, 2020. Accessed: 2024-05-19.

[5] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 934–950. Springer, 2013.

[6] Simon Alford, Anshula Gandhi, Akshay Rangamani, Andrzej Banburski, Tony Wang, Sylee Dandekar, John Chin, Tomaso Poggio, and Peter Chin. Neural-guided, bidirectional program search for abstraction and reasoning. In *Complex Networks & Their Applications X: Volume 1, Proceedings of the Tenth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2021 10*, pages 657–668. Springer, 2022.

[7] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[8] Ethem Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 3 edition, 2014.

[9] Rajeev Alur, Rastislav Bodík, 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 *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[10] Rajeev Alur, Rastislav Bodík, 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 *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 1–8, 2013.

[11] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[12] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

[13] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[14] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, November 2018.

[15] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

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

[17] R. Harald Baayen, Richard Piepenbrock, and H van Rijn. The CELEX lexical database on CD-ROM, 1993.

[18] Xuefeng Bai, Jialong Wu, Yulong Chen, Zhongqing Wang, and Yue Zhang. Constituency parsing using llms. *arXiv preprint arXiv:2310.19462*, 2023.

[19] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[20] Andrzej Banburski, Anshula Gandhi, Simon Alford, Sylee Dandekar, Sang Chin, and tomaso a poggio. Dreaming with ARC. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

[21] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[22] Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *arXiv preprint arXiv:2309.12288*, 2023.

[23] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[24] 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, 2015.

[25] Paul Boersma and Bruce Hayes. Empirical Tests of the Gradual Learning Algorithm. *Linguistic Inquiry*, 32(1):45–86, 2001.

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

[27] Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W Zhang, Michaël Defferrard, and Taco Cohen. Codeit: Abstract reasoning with iterative policy-guided program synthesis. 2023.

[28] Donal E Carlston. *Dual-Process Theories*. 2013.

[29] Jeff Carver. The impact of background and experience on software inspections. *Empirical Softw. Engg.*, 9(3):259–262, sep 2004.

[30] Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics*, 2:491–504, 2014.

[31] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 963–975, New York, NY, USA, 2018. Association for Computing Machinery.

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

[33] 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.

[34] Qiaochu Chen, Shankara Pailoor, Celeste Barnaby, Abby Criswell, Chenglong Wang, Greg Durrett, and Işil Dillig. Type-directed synthesis of visualizations from natural language queries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):532–559, 2022.

[35] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.

[36] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[37] Noam Chomsky and Morris Halle. *The Sound Pattern of English*. Studies in language. Harper & Row, 1968.

[38] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek B Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Oliveira Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

[39] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP'00, pages 268–279. Association for Computing Machinery, Sep 2000.

[40] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[41] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[42] Gabriel Doyle, Klinton Bicknell, and Roger Levy. Nonparametric Learning of Phonological Constraints in Optimality Theory. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 1094–1103, 2014.

[43] Gabriel Doyle and Roger Levy. Data-driven learning of symbolic constraints for a log-linear model in a phonological setting. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 2217–2226, 2016.

[44] Michael Droettboom, Roman Yurchak, Hood Chatham, Dexter Chua, Gyeongjae Choi, Marc Abramowitz, casatir, Jan Max Meyer, Jason Stafford, Madhur Tandon, Michael

Greminger, Grimmer Kang, Chris Trevino, Wei Ouyang, Joe Marshall, Adam Seering, Nicolas Ollinger, Ondřej Staněk, Sergio, Teon L Brooks, Jay Harris, Alexey Ignatiev, Seungmin Kim, Paul m. p. P., jcaesar, Carol Willing, Cyrille Bogaert, Dorian Pula, Frithjof, and Michael Jurasovic. Pyodide: A Python distribution for WebAssembly (0.19.0), January 2022.

[45] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12, 2020.

[46] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. *Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists*, pages 1–12. Association for Computing Machinery, New York, NY, USA, 2020.

[47] Kevin Ellis, Adam Albright, Armando Solar-Lezama, Joshua B. Tenenbaum, and Timothy J. O'Donnell. Synthesizing theories of human language with Bayesian program induction. In Prep, 2019.

[48] Kevin Ellis, Adam Albright, Armando Solar-Lezama, Joshua B Tenenbaum, and Timothy J O'Donnell. Synthesizing theories of human language with bayesian program induction. *Nature communications*, 13(1):5024, 2022.

[49] Kevin Ellis, Lucas Morales, Mathias Sablé Meyer, Armando Solar-Lezama, and Joshua B Tenenbaum. Search, compress, compile: Library learning in neurally-guided bayesian program learning. *Advances in neural information processing systems*, 2018.

[50] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

[51] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018.

[52] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in neural information processing systems*, pages 973–981, 2015.

[53] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[54] 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.

[55] 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, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

[56] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. *ACM SIGPLAN Notices*, 52(1):599–612, 2017.

[57] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *POPL*, 2017.

[58] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Loopy: Interactive program synthesis with control structures. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[59] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 614–626, New York, NY, USA, 2020. Association for Computing Machinery.

[60] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[61] Raphael Fischer, Matthias Jakobs, Sascha Mücke, and Katharina Morik. Solving abstract reasoning tasks with grammatical evolution. In *LWDA*, pages 6–10, 2020.

[62] 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, 2016. ACM.

[63] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[64] Nat Friedman, Jun 2021.

[65] Richard Futrell, Adam Albright, Peter Graff, and Timothy J. O'Donnell. A generative model of phonotactics. *Transactions of the Association for Computational Linguistics*, 5:73–86, 2017.

[66] Jianhang Gao, Qing Zhao, Wei Ren, Ananthram Swami, Ram Ramanathan, and Amotz Bar-Noy. Dynamic shortest path algorithms for hypergraphs. *CoRR*, abs/1202.0082, 2012.

[67] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[68] Carlos Gemmell and Jeffrey Dalton. Generate, transform, answer: Question specific tool synthesis for tabular data. *arXiv preprint arXiv:2303.10138*, 2023.

[69] Daniel Gildea and Daniel Jurafsky. Learning bias and phonological-rule induction. *Computational Linguistics*, 22(4):497–530, 1996.

[70] Barney G. Glaser and Anselm L. Strauss. *The discovery of grounded theory: strategies for qualitative research*. Aldine Transaction, 5. paperback print edition, 1967.

[71] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.*, 22(2), mar 2015.

[72] John Goldsmith and Jason Riggle. Information theoretic approaches to phonological structure: the case of Finnish vowel harmony. *Natural Language & Linguistic Theory*, 30(3):859–896, 2012.

[73] Sharon Goldwater and Mark Johnson. Learning OT Constraint Rankings Using a Maximum Entropy Model. *Proceedings of the Stockholm Workshop on Variation within Optimality Theory*, pages 111–120, 2003.

[74] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[75] 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, 2011. ACM.

[76] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.

[77] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[78] Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. Learning to complete code with sketches. In *International Conference on Learning Representations*, 2021.

[79] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[80] Carlos Gussenhoven and Haike Jacobs. *Understanding Phonology*. Routledge, 2017.

[81] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[82] Bruce Hayes and Colin Wilson. A maximum entropy model of phonotactics and phonotactic learning. *Linguistic Inquiry*, 39(3):379–440, 2008.

[83] Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. Tabllm: Few-shot classification of tabular data with large language models. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent, editors, *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 5549–5581. PMLR, 25–27 Apr 2023.

[84] Brian Hempel and Ravi Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th annual symposium on user interface software and technology*, pages 379–390, 2016.

[85] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *ArXiv*, abs/2105.09938, 2021.

[86] Jeevana Priya Inala and Rishabh Singh. WebRelate: integrating web data with spreadsheets using examples. *PACMPL*, 2(POPL):2:1–2:28, 2018.

[87] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *International Conference on Software Engineering (ICSE)*, May 2022.

[88] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: Synthesis-aided api discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[89] Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. Exploring the learnability of program synthesizers by novice programmers. page 15, 2022.

[90] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[91] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. Discovering the syntax and strategies of natural language programming with generative language models. In *CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2022.

[92] Martin Josifoski, Marija Sakota, Maxime Peyrard, and Robert West. Exploiting asymmetry for synthetic training data generation: Synthie and the case of information extraction. *arXiv preprint arXiv:2303.04132*, 2023.

[93] Daniel Kahneman. *Thinking, fast and slow*. Penguin psychology. Penguin Books, 2011.

[94] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[95] Kite. Kite: Ai-powered completions for jupyterlab. https://www.kite.com/integrations/jupyter/, 2020.

[96] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. *SIGPLAN Not.*, 48(10):407–426, October 2013.

[97] Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. An update on deductive synthesis and repair in the leon tool. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, pages 100–111, 2016.

[98] Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. On repair with probabilistic attribute grammars. *arXiv preprint arXiv:1707.04148*, 2017.

[99] Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

[100] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, 2017.

[101] Seungpil Lee, Woochang Sim, Donghyeon Shin, Sanha Hwang, Wongyu Seo, Jiwon Park, Seokki Lee, Sejin Kim, and Sundong Kim. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus, 2024.

[102] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

[103] Géraldine Legendre, Yoshiro Miyata, and Paul Smolensky. Harmonic Grammar – A formal multi-level connectionist theory of linguistic well-formedness: Theoretical foundations. Technical Report ICS # 90-5, CU-CS-465-90, University of Colorado, 1990.

[104] Chao Lei, Nir Lipovetzky, and Krista A. Ehinger. Generalized planning for the abstraction and reasoning corpus, 2024.

[105] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7. ACM, Apr 2020.

[106] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. Efficient bottom-up synthesis for programs with local variables. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[107] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models, 2024.

[108] 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, 2022.

[109] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584*, 2020.

[110] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[111] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[112] Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, Oct 2021.

[113] Zohar Manna and Richard Waldinger. Synthesis: dreams→ programs. *IEEE Transactions on Software Engineering*, (4):294–328, 1979.

[114] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, apr 1960.

[115] R Thomas McCoy, Shunyu Yao, Dan Friedman, Matthew Hardy, and Thomas L Griffiths. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*, 2023.

[116] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

[117] Smitha Milli, Falk Lieder, and Thomas L. Griffiths. A rational reinterpretation of dual-process theories. *Cognition*, 217:104881, 2021.

[118] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[119] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.

[120] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. recode: A lightweight find-and-replace interaction in the ide for transforming code by example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 258–269, 2021.

[121] David Odden. *Introducing Phonology*. Cambridge University Press, 2005.

[122] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

[123] OpenAI. Hello gpt-4.0, 2024. Accessed: 2024-05-19.

[124] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[125] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. Flashprofile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[126] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. Programming with a read-eval-synth loop. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[127] Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming, ECOOP*, 2020.

[128] 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. ACM, 2018. tex.ids: pelegProgrammingNotOnly2018a event-place: Gothenburg, Sweden.

[129] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, Jul 1987.

[130] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *ACM Sigplan Notices*, 49(6):408–418, 2014.

[131] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. *SIGARCH Comput. Archit. News*, 44(2):297–310, March 2016.

[132] Alan Prince and Paul Smolensky. *Optimality Theory: Constraint interaction in generative grammar*. Wiley-Blackwell, 2004.

[133] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models. *ArXiv*, abs/2204.00498, 2022.

[134] Ezer Rasin, Iddo Berger, Nur Lan, and Roni Katzir. Acquiring opaque phonological interactions using Minimum Description Length. In *Supplemental Proceedings of the 2017 Annual Meeting on Phonology*, 2017.

[135] 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*, pages 419–428, 2014.

[136] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*, pages 74–83. Springer, 2019.

[137] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2021.

[138] Iggy Roca and Wyn Johnson. *A Workbook in Phonology*. Blackwell, 1999.

[139] Advait Sarkar. Is explainable AI a race against model complexity? *arXiv preprint arXiv:2205.10119*, 2022.

[140] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213*, 2022.

[141] Advait Sarkar, Mateja Jamnik, Alan F Blackwell, and Martin Spott. Interactive visual machine learning in spreadsheets. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 159–163. IEEE, 2015.

[142] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[143] Rohin Shah, Sumith Kulal, and Rastislav Bodik. Scalable synthesis with symbolic syntax graphs. 2018.

[144] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Trans. Program. Lang. Syst.*, 44(2), may 2022.

[145] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

[146] Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[147] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis, 2019.

[148] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[149] Calvin Smith and Aws Albarghouthi. Program synthesis with equivalence reduction. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 24–47, 2019.

[150] Armando Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[151] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[152] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

[153] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 120–131. Association for Computing Machinery, May 2016.

[154] Anselm L. Strauss and Juliet Corbin. *Basics of Qualitative Reseach: Grounded Theory Procedures and Techniques*. SAGE Publications, Inc., 1990.

[155] TabNine. Tabnine: Ai assistant for development teams. https://www.tabnine.com/, 2018.

[156] John Chong Min Tan and Mehul Motani. Large language model (llm) as a system of multiple expert agents: An approach to solve the abstraction and reasoning corpus (arc) challenge. *arXiv preprint arXiv:2310.05146*, 2023.

[157] Bruce Tesar and Paul Smolensky. *Learnability in Optimality Theory*. MIT Press, 2000.

[158] Immanuel Trummer. Codexdb: Generating code for processing sql queries using gpt-3 codex. *ArXiv*, abs/2204.08941, 2022.

[159] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

[160] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Improving llm code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*, 2024.

[161] Priyan Vaithilingam, Tianyi Zhang, and Elena Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Late-Breaking Work*, 2022.

[162] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. (arXiv:1706.03762), Dec 2017. arXiv:1706.03762 [cs].

[163] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5052–5061, 2018.

[164] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 382–394, 2022.

[165] 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*, pages 452–466. ACM, 2017.

[166] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[167] Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D Goodman. Hypothesis search: Inductive reasoning with language models. *arXiv preprint arXiv:2309.05660*, 2023.

[168] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. volume 2, pages 63:1–63:30, New York, NY, USA, December 2017. ACM.

[169] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. volume 1, pages 62:1–62:26, New York, NY, USA, October 2017. ACM.

[170] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[171] Eric Wastl. Advent of code. https://adventofcode.com/2021, 2021.

[172] R. L Weide. The CMU pronouncing dictionary. Release 0.7b, 2014.

[173] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. Perfection not required? human-ai partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*, pages 402–412, New York, NY, USA, 2021. Association for Computing Machinery.

[174] Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. Grounding data science code generation with input-output specifications, 2024.

[175] Jack Williams, Carina Negreanu, Andrew D Gordon, and Advait Sarkar. Understanding and inferring units in spreadsheets. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE Computer Society, 2020.

[176] Johan Sokrates Wind. Arc kaggle competition, 1st place, 2020. Accessed: 2024-05-19.

[177] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation, 2020.

[178] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges, 2021.

[179] Yudong Xu, Elias B Khalil, and Scott Sanner. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4115–4122, 2023.

[180] Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*, 2023.

[181] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[182] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 174–184, 2023.

[183] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics, 2018.

[184] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: Continual pre-training on sketches for library-oriented code generation. *ArXiv*, abs/2206.06888, 2022.

[185] Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pages 40932–40945. PMLR, 2023.

[186] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. Interpretable program synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[187] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, New York, NY, USA, 2020. Association for Computing Machinery.

[188] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017.

[189] Xiangyu Zhou, Ras Bodik, Alvin Cheung, and Chenglong Wang. Synthesizing analytical sql queries from computation demonstration. In *PLDI*, 2022.