Reverse Sketching

Tyler Holloway [1], Chelse Swoopes [1], Ian Arawjo [1], Hila Peleg [2] and Elena Glassman [1]

[1] *Harvard University, Cambridge, MA*
[2] *Technion, Haifa, ISR*

Abstract

To avoid overwhelming users, many interactive PBE systems only show the highest-ranked programs, even when the system generates tens or hundreds of programs that satisfy the provided examples. Although this approach can significantly reduce the number of candidate programs that a user must consider, it limits their ability to fully understand the solution space and choose the best program for their specific needs. We hypothesize that offering users an overview of the entire space of solutions will help them identify the desired program, even if it is not among the highest ranked ones; identify other, potentially better programs that satisfy the specification; and provide insight into how the synthesizer behaves based on the provided examples, which may help users provide better examples in the future. In this paper, we introduce reverse sketching, a novel algorithm that uses anti-unification to generate a set of sketches, or partially-complete programs with 'holes', that capture the syntactic structure of multiple programs in the solution space.

Keywords: Program Synthesis. Programming by Example. Human-Computer Interaction.

1 Introduction

Programming can be a challenging task for novice programmers due to the complex nature of programming languages and the various tools required to develop, test, and deploy programs. Programming-by-Example (PBE) is the task of automatically generating a computer program that satisfies a set of examples. The examples describe the desired program behavior, and they can take a variety of forms, including input-output pairs [1]–[3], partially-complete programs [4], [5], and natural language instructions [6]. The mass market success of inductive-synthesis-powered tools, such as FlashFill [7] and FlashExtract [8], suggests that PBE has the potential to allow people with little or no programming experience to author their own programs by example.

While relatively easy to provide, examples are only a partial specification of the user's intent. As a result, many PBE engines can generate tens, hundreds, and sometimes hundreds of thousands of candidate programs that meet this partial specification [9]. For example, the FlashFill system generates over a million candidate programs from the input-output example "John Doe" -> "J. Doe" [7]. Several interactive PBE systems only show the highest-ranked candidate programs [1], [3], [10], [11]. The ranks of the candidates are determined by a ranking function, which can be based on a variety of factors, such as hand-crafted heuristics [7], hard-coded distributions [12], and probabilistic models trained on large code corpora [13]. Although this approach can significantly reduce the number of candidate programs a user must consider, it limits their ability to fully understand the solution space and choose the best program for their specific needs.

This paper introduces reverse sketching, a novel algorithm that uses anti-unification to generate a set of sketches, or partially complete programs with 'holes'. Each sketch captures the syntactic structure of a subset of programs in the solution space, and the holes signify the syntactic locations where the programs in the subset vary. We demonstrate the effectiveness of this technique on string manipulation tasks, a popular task domain for PBE synthesizers [7]. In future work, we plan to develop an interface that allows users to interact with the sketches. Additionally, we will conduct a user study to test the following hypotheses: We hypothesize that offering users an overview of the entire space of solutions will help them identify the desired program, even if it's not among the highest ranked programs; identify other potentially better e.g.,more robust, programs that satisfy the specification that have not yet been synthesized; and provide insight into how the synthesizer behaves based on the provided examples, which may help users provide better examples in the future.

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

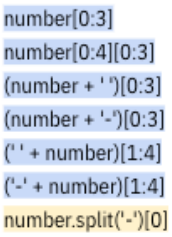

Figure 1. A subset of the candidate programs partitioned by root node type.

2 Related Work

Programming by example (PBE) is a subfield of program synthesis that automatically generates computer programs that satisfy a set of examples. Traditionally, PBE tools for textual languages operate using "sketches," which are partially completed program with "holes" [4]. The term "sketch" comes from the notion of program sketching, where the user provides a partially-complete program that outlines the desired implementation strategy and delegates the low-level details to a synthesizer [4]. In recent years, sketches have proven to be a versatile tool; for example, Galenson et al. leverage user-provided sketches to refine the list of synthesized programs [14], and Zhang et al. use sketches to summarize programs generated by the synthesizer at a specific point in the search process [15].

The mass market success of tools powered by inductive synthesis, such as FlashFill [7] and FlashExtract [8], shows that PBE can allow people in the real world to author programs in specific domains by example rather than by directly writing code. Yet, despite this initial commercial success, PBE systems present at least three usability challenges: First, there is a "user-synthesizer gap," or a mismatch between the user's mental model of the synthesizer's capabilities and its actual capabilities [2]. Mental models influence a users' perception of a system, their ability to understand whether and how a system can be used for particular use cases, and even their decision to adopt it [16]. Second, providing examples that are unambiguous and completely describe the desired program behavior is difficult, which is exacerbated by the observation that PBE users are often reluctant to provide more than a few illustrative examples [9], [17]. Third, evaluating the system's output can be challenging, as a PBE system can generate zero, tens, hundreds, and sometimes thousands of candidate programs that satisfy the user's partial specification [9]. In fact, for common programming languages such as Python and C++, the space of consistent programs can be extremely large, even infinite [9]. To address this challenge, many PBE systems adopt a 'top-K' approach, where only the K highest-ranked candidate programs are presented to the user [3], [10]. Recent ML-powered systems like GitHub CoPilot go further, only presenting the single most highly ranked program and eliminating user choice entirely. Top-k approaches can be effective in reducing the solution space that the user can consider but they may also exacerbate the user-synthesizer gap and the example refinement processes mentioned above, limiting the user's ability to fully explore the solution space and choose the best program for their specific needs.

There are few interfaces that facilitate user exploration of the solution space beyond the top-K. Mayer et al. proposed a novel interaction model called Program Navigation, which allows users to look beyond the top-K-ranked solutions and explore the possible sub-expressions of the top-ranked program [11]. The navigational interface accomplishes this by taking advantage of the fact that candidate programs share common sub-expressions. The approach, however, fails to take advantage of the fact that candidate programs often share more than common sub-expressions; they often share common high-level syntactic structures [11]. In this paper, we use program sketches to exploit this commonality. Our tool operates by partitioning the set of solutions into subsets of syntactically similar programs and capturing the shared syntactic structure of each subset with a sketch.

3 Reverse Sketching Algorithm

In this section, we introduce reverse sketching, an algorithm that uses anti-unification to generate a collection of sketches, each of which captures the syntactic structure of multiple programs. More

```
number[0:3]
number.split('-')[0]
(__ + __)[0:3]
(__ + __)[1:4]
number[0:4][0:3]
(number + ___).split('-')[0]
(__ + number).split('-')[1]
number.replace(___, ___)[0:3]
```

Figure 2. The sketches for extracting the area code from a U.S. phone number from 70 synthesis-generated programs.

precisely, anti-unification is the task of creating the most specific generalization, or anti-unifier, of a set of terms. For a set of terms $t_1, t_2, \ldots, t_n$, the anti-unifier t is a pattern from which the terms $t_1, t_2, \ldots, t_n$ can be obtained by substitution of t's variables. A substitution is a mapping $\sigma : V \to T$ from variables to terms, where $\{x_1 \to t_1, \ldots, x_k \to t_k\}$ represents the substitutions for each variable x_i to the term t_i, for $i = 1, \ldots, k$. Applying the substitution to a term t replaces every occurrence of each variable x_i in the term t with t_i.

The algorithm accepts a set $\mathcal{T}$ of abstract syntax trees (ASTs) of all candidate programs. It then employs anti-unification to generate a set of sketches. Even though anti-unifying every $T \subseteq \mathcal{T}$ computes the entire space of sketches, there are $2^{\mathcal{T}}$ such subsets, making this method impractical in terms of memory and compute time. Our approach overcomes this challenge by dividing the algorithm into four stages. First, we partition $\mathcal{T}$ into sets $\mathcal{T}_r$ such that two trees are in the same partition if they both have the root node r. For example, both `1 + arg` and `arg + 3` will be in $\mathcal{T}_+$. Partitioning is based on the intuition that ASTs with similar root nodes are likely to have similar structures and perform similar operations, and it also reduces the space for anti-unification as the anti-unification of each $\mathcal{T}_r$ will have r as its root, whereas $\mathcal{T}_{r_1} \cup \mathcal{T}_{r_2}$ will have a new variable at its root.

In the second stage, the algorithm generates sketches for each $\mathcal{T}_r$. The resulting sketch captures the common structure and operations performed by the ASTs in $\mathcal{T}_r$, and will serve as a representation of the ASTs in each cluster. In the third stage, the algorithm further groups the ASTs in each $\mathcal{T}_r$ by the types of their substitution values. A substitution value is a value that replaces a variable in the sketch. By grouping the ASTs based on substitution values, we can further reduce the search space for anti-unification, as we only need to consider sketches with the same substitution values, and generate more specific sketches. Finally, the members of the groups produced in step 3 are anti-unified. These are the final sketches shown to the user.

In summary, the algorithm starts with a set of programs, divides them into groups based on their syntax, performs anti-unification to generate sketches for each group, partitions them further based on their substitution mappings, and finally generates more specific sketches.

Consider the set of programs presented in Figure 1. The algorithm partitions these programs into two groups. The first set includes all programs with a blue background, while the second set contains a single program with a yellow background. In the next stage, the algorithm performs anti-unification on both groups, producing two sketches. The sketch for the first set, denoted as G_1, is `__[__:__]`, which represents a prettified view of their anti-unifier $x_1[x_2 : x_3]$, with every variable x_i replaced with `'__'`. The substitution mappings σ_1, σ_2, and so on, are defined such that $\sigma_1 = \{x_1 \to \texttt{number}, x_2 \to 0, x_3 \to 3\}$, $\sigma_2 = \{x_1 \to \texttt{number[0:4]}, x_2 \to 0, x_3 \to 3\}$, and so forth.

```
row[2].split('@')[0]
(row[2] + ___).split('@')[0]
(___[0:1] + row[1].lower)
(___[0:1] + row[1]).lower
(row[3] + row[___]).lower().split('@')[0]
row[0][0:1].lower() + row[1].lower()
(row[0][0:1] + row[1].lower()).lower()
(row[0][0:1].lower() + row[1]).lower()
row[2][0:len(row[2].split('@')[0])]
```

Figure 3. The sketches for extracting email name from an excel-sheet-like row containing 3 columns: [First name, Last name, Email].

On the other hand, the second set G_2 consists of a single program, and so its anti-unification has no substitutions and is simply the full program `number.split('-')[0]`. In the third stage, the sets generated in the previous step are partitioned based on the types and placement of constants in their substitution mappings. For example, G_1 is divided into two subsets: {`number[0:3]`} and {`number[0:4][0:3]`, `(number + '-')[0:4][0:3]`}. Finally, the algorithm anti-unifies these new sets, resulting in the generation of two more specific sketches: `number[0:3]` and `_[0:4][0:3]`.

The results of this algorithm are shown in Figures 4 and 5 on two string manipulation tasks: area code extraction and email name extraction. Although this paper targets the Python language, we believe that it can be used to generate reverse sketches for other programming languages since anti-unification operates on a syntactic level. One of the main benefits of anti-unification is that it allows for the identification of common structures or patterns across different programs, even when they have different variable names or are expressed in different ways.

4 Future Work

The next step of this work is to develop an interface that allows the user to easily explore and compare the concrete programs that fall under the sketches generated by our *reverse sketching* algorithm. To evaluate the performance and usability of our proposed model, we will conduct a within-subjects controlled lab study to examine participants' ability to synthesize the correct program with a PBE-powered interface that shows the top-k candidate programs compared to a PBE-powered interface that shows the entire space of solutions organized by reverse sketches.

4.1 Participants

We will recruit 15-25 participants who have a minimum of 1 year of experience programming in Python, with varying levels of expertise in writing, debugging, and testing programs that perform string manipulation tasks. The participants will be recruited from a computer science department of at least one research university in the United States, and we will ensure that they represent a diverse range of programming experience, age, gender, and ethnicity. For example, some participants may be experienced in using string methods like `split()`, `join()`, and `replace()`, while others may be less experienced in these tasks. All participants will receive compensation for their time, in the form of a $25 gift card.

4.2 Protocol

Prior to the study, the participants will complete a pre-study survey, which will gather information on their programming experience, familiarity with Python, and previous exposure to PBE systems. They will also be provided with a study consent form to grant or deny permission for audio and video recording of the session. To introduce the system, we will provide participants with a brief tutorial on the PBE-powered interface, including how to use the interface to generate and refine programs. The tutorial will also cover the basics of string manipulation tasks in Python, including the relevant syntax and commonly used functions. Participants will be randomly assigned to complete two string manipulation tasks using the PBE-powered interface, one using the top-K candidate program display and the other using the entire space of programs display. The tasks will be designed to require participants to use a variety of string manipulation techniques and functions. During each task, participants will be given a prompt and asked to write a program that fulfills the prompt using the PBE-powered interface. Each task will have a time limit of 30 minutes to ensure that participants are able to complete the tasks within a reasonable timeframe. Participants will be asked to "think aloud" as they carry out the tasks. After completing each task, participants will be asked to (i) complete a survey to reflect on the effectiveness and usability of the interface they used and (ii) take part in a semi-structured interview in which they will discuss their thoughts on the interface. We will record the participants' interactions with the interface to analyze their programming behavior and performance. At the end of the experiment, we will compare the participants' ability to synthesize the correct program using the two different display methods, as well as their subjective experience and feedback. The within-subjects design ensures that each participant will experience both interface

display methods, reducing the impact of individual differences on the study results. By comparing the performance and feedback of participants across both methods, we can evaluate the effectiveness of each display method for PBE-powered interfaces and gain insights into how to improve the design of such interfaces in the future.

4.3 Tasks

Input	Output
456-281-2822	456
896-261-2182	896

Figure 4. Input-output examples for area code extraction.

Input	Output
[Bobby, Smith, bsmith@aol.com]	bsmith
[Alex, Rolls, arolls@aol.com]	arolls

Figure 5. Input-output examples for email name extraction.

The participants will be tasked with completing two string-manipulation tasks using the Python programming language. Examples of such tasks are area code and email name extraction, as shown in Figure 4 and Figure 5. Tasks will be taken from Leetcode, a popular Question Pool website (QP) that includes a variety of coding questions on different topics, such as dynamic programming, sorting, and searching.

5 Discussion

In this paper, we introduced *reverse sketching*, a novel algorithm that generates a set of sketches that capture the syntactic structure of multiple programs in the solution space. As the next step, we will develop an interface that allows users to explore the solution space by interacting with the sketches and evaluate its effectiveness through user studies. Future work may involve extending and generalizing the algorithm to handle different programming languages, data structures, and programming paradigms. Another potential avenue for future research is exploring the algorithm's effectiveness in various domains and contexts. However, it should be noted that adapting the algorithm to different contexts may pose challenges, particularly in more complex and abstract domains that require additional knowledge or expertise. Furthermore, future work could investigate integrating machine learning and natural language processing techniques into reverse sketching to enhance its adaptability and user experience.

6 Acknowledgements

This work was supported in part by the first author's NSF Graduate Research Fellowship and the NSF Grants 2107391 and 2123965. Any opinions, findings or recommendations expressed here are those of the authors and do not necessarily reflect views of the sponsors.

References

[1] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 627–648, ISBN: 9781450375146. DOI: 10.1145/3379337.3415900. [Online]. Available: https://doi.org/10.1145/3379337.3415900.

[2] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, "Small-step live programming by example," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 614–626.

[3] H. Peleg and N. Polikarpova, "Perfect is the enemy of good: Best-effort program synthesis," *Leibniz international proceedings in informatics*, vol. 166, 2020.

[4] A. Solar-Lezama, "The sketching approach to program synthesis," in *Asian Symposium on Programming Languages and Systems*, Springer, 2009, pp. 4–13.

[5] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 497–518, 2013.

[6] A. Desai, S. Gulwani, V. Hingorani, *et al.*, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 345–356, ISBN: 9781450339001. DOI: 10.1145/2884781.2884786. [Online]. Available: https://doi.org/10.1145/2884781.2884786.

[7] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *SIGPLAN Not.*, vol. 46, no. 1, pp. 317–330, Jan. 2011, ISSN: 0362-1340. DOI: 10.1145/1925844.1926423. [Online]. Available: https://doi.org/10.1145/1925844.1926423.

[8] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," *SIGPLAN Not.*, vol. 49, no. 6, pp. 542–553, Jun. 2014, ISSN: 0362-1340. DOI: 10.1145/2666356.2594333. [Online]. Available: https://doi.org/10.1145/2666356.2594333.

[9] S. Gulwani, O. Polozov, R. Singh, *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[10] J. Hu, P. Vaithilingam, S. Chong, M. Seltzer, and E. L. Glassman, "Assuage: Assembly synthesis using a guided exploration," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 134–148, ISBN: 9781450386357. DOI: 10.1145/3472749.3474740. [Online]. Available: https://doi.org/10.1145/3472749.3474740.

[11] M. Mayer, G. Soares, M. Grechkin, *et al.*, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software Technology*, ser. UIST '15, Charlotte, NC, USA: Association for Computing Machinery, 2015, pp. 291–301, ISBN: 9781450337793. DOI: 10.1145/2807442.2807459. [Online]. Available: https://doi.org/10.1145/2807442.2807459.

[12] K. Ellis, A. Solar-Lezama, and J. B. Tenenbaum, "Sampling for bayesian program learning," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, Barcelona, Spain: Curran Associates Inc., 2016, pp. 1297–1305, ISBN: 9781510838819.

[13] S. Handa and M. C. Rinard, "Inductive program synthesis over noisy data," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 87–98.

[14] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 653–663.

[15] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. L. Glassman, "Interpretable program synthesis," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–16.

[16] D. A. Norman, "Some observations on mental models," in *Mental models*, Psychology Press, 2014, pp. 15–22.

[17] T. Y. Lee, C. Dugan, and B. B. Bederson, "Towards understanding human mistakes of programming by example: An online user study," in *Proceedings of the 22nd International Conference on Intelligent User Interfaces*, ser. IUI '17, Limassol, Cyprus: Association for Computing Machinery, 2017, pp. 257–261, ISBN: 9781450343480. DOI: 10.1145/3025171.3025203. [Online]. Available: https://doi.org/10.1145/3025171.3025203.