Human-Centric Program Synthesis

Will Crichton
Stanford University
wcrichto@cs.stanford.edu

Abstract

Program synthesis techniques offer significant new capabilities in searching for programs that satisfy high-level specifications. While synthesis has been thoroughly explored for input/output pair specifications (programming-by-example), this paper asks: what does program synthesis look like beyond examples? What actual issues in day-to-day development would stand to benefit the most from synthesis? How can a human-centric perspective inform the exploration of alternative specification languages for synthesis? I sketch a human-centric vision for program synthesis where programmers explore and learn languages and APIs aided by a synthesis tool.

2012 ACM Subject Classification Software and its engineering → Programming by example

Keywords and phrases Program synthesis, programming by example, PL/HCI

1 A Story of Our Time

Consider the story of Dana the Data Scientist. At Sonmanto, her agritech business, Dana wants to analyze the weekly seed production. Opening a Jupyter notebook, she creates a new code cell, imports a few libraries, and sends off a SQL query to build a Pandas dataframe. Knowing the Pandas API from her data science course and past experience, she computes the week's average seed production using standard dataframe methods.

```
query = sql("SELECT time, production FROM seed_production ORDER BY time")
df = pd.read_sql_query(query)
df.where(df.time >= dt.now() - dt.timedelta(days=7)).production.mean()
```

Concerned that the week's production seems low, Dana wants to see a 7-day rolling average of the last year's production to put this week into context. She has never computed a weekly rolling average before, so she Googles "pandas rolling average". Excellent, Pandas has a `Dataframe.rolling` method, but... it doesn't do quite what she wants. All the examples use windows that contains a fixed number of elements, but she wants windows of a fixed duration potentially containing different numbers of production samples.

Dana continues searching increasingly elaborate queries like "pandas rolling average date dynamic window", and eventually finds some StackOverflow answers that look almost right. However, all of their solutions either use abstract notation like "foobar" or were made for other domains like stock trading. Dana finds it difficult to see the relationship between finance problems and seed production. After twenty minutes of searching, she gives up with a resigned sigh and decides to just implement it in plain Python.

```
for day_start in pd.date_range(df.time.min(), df.time.max()):
    day_end = day_start + datetime.timedelta(days=7)
    window = [row.production
              for day in pd.date_range(day_start, day_end)
              for _, row in df.iterrows() if row.time.date() == day.date()]
    weekly_prod.append(pd.Series(window).mean())
```

Dana knows the code isn't beautiful, but it gets the job done. Glancing back at the StackOverflow questions, she starts to see the connections after going through her own

implementation. But it's close to lunch, and she spent too long on this already. Simplifying the code is a task for another day, and she moves on.

2 A Story of Another Time

. . . Concerned that the week's production seems low, Dana wants to see a 7-day rolling average of the last year's production to put this week into context. She has never computed a weekly rolling average before, so she Googles "pandas rolling average". Excellent, Pandas has a `Dataframe.rolling` method, but. . . it doesn't do quite what she wants.

Rather than continuing to search fruitlessly, she writes down her plain Python solution. Dana highlights the code cell and clicks "Synthesize" in her Jupyter toolbar, opening a dialog box on the side. She writes `Dataframe.rolling` and `Dataframe.mean` into the box, knowing those are likely going to be important parts of a Pandas-specific solution if it exists. Guided by her suggestions, the synthesis engine finishes in under a minute, producing a `rolling` solution contextualized to her dataframe.

```
df.sort_values('time').set_index('time').rolling('7d').mean()
```

Ahh, the rolling function has a special syntax for time windows. But, Dana wonders, what does each part do? Hovering over each part of the program, the synthesis tool uses its counterexamples to shows what would happen if a given method call was omitted or changed. Removing `sort_values` or `set_index` cause the program to raise an error. Changing the window to `rolling(7)` produces an incorrect output.

Plotting the values in Matplotlib, Dana marvels at the simplicity of the solution. She starts to wonder: are there other places in Sonmanto's code base where they could use this pattern? Glancing over at the clock, there's still an hour to lunch, great! Highlighting her old code cell once more, she clicks "Find Similar" to search her notebooks and text files for snippets that look structurally similar to the one she just wrote.

After the search engine returns five plausibly similar programs, Dana runs the synthesis engine in parallel on each one. Noticing that most of the snippets were written by Danny the Data Engineer, she motions Danny over and teaches him about the feature she just learned.

3 The Past and Present of Program Synthesis

The stories above highlight a key fact about modern-day programming: programmers routinely deal with dozens of representations of code and data. In the data science domain, Jupyter notebooks swap between explanation and code. Data flows from SQL databases to Python lists to Pandas dataframes. Operations mix and match bespoke APIs with general-purpose programming constructs. Programmers are continually learning new representations as languages, libraries, and tools emerge and change.

Dana's struggles with Pandas show a prototypical case of acquiring a new representation. Not knowing how to compute a specific kind of rolling average, she uses a combination of documentation, code examples, and prior knowledge to understand whether the Pandas API can solve her problem. Having general programming skills, she can arrive at a standard Python solution, but not the more concise API-specific solution. As the second story demonstrates, I believe that program synthesis techniques hold promise in helping programmers overcome these kinds of representational transfer problems (or refactoring, migration, etc.). Yet, to date, such a story is still a fantasy.

To understand why, we will briefly examine the history of program synthesis. Synthesis has been predominantly applied in the context of programming-by-example (PBE). In PBE, a

user provides examples (input/output pairs) of a pure function, and the synthesizer attempts to find a "good" (e.g. small) function that satisfies those examples. Often, the user is an end-user manipulating spreadsheets or text documents, and the generated program is an invisible macro. Through hard-earned experience with dozens of PBE systems, researchers have both articulated design principles of PBE [8] and ultimately produced the flagship commercial synthesis engine, Excel FlashFill [11]. This effort succeeded in part by a human-centric push to understand both the applications where PBE was most valuable, and the essential usability constraints for real-world usage.

The central question of this paper, then: what does human-centric synthesis look like beyond PBE? Specifically, what applications open up when a user has the programming skills to express specifications at a level beyond examples? Traditionally, these kinds of tasks have been viewed as refactoring or migration, where the existing codebase specifies the desired behavior for the transformed one. Historically, refactoring tools could only perform simple syntactic changes like renaming types or methods. However, recent synthesis tools have shown striking progress in translating between complex high-level representations of code. For example, tools can move between languages like Java → Spark [1], and Fortran → Halide [5]. Tools can refactor APIs, like parallelizing Java streams [7], adding default methods in Java [6], updating SQL queries after schema changes [14].

These approaches have significantly advanced the state-of-the-art in program synthesis techniques. But given the lack of meaningful commercial adoption, it is unclear whether they're trying to solve the right problem. Certainly this fact arises in part from the general difficulty of tech transfer. But this would not be the first time the PL and compilers communities have been led astray by the allure of automating high-level tasks for programmers.

For example, modern compilers do register allocation by solving a complex graph coloring problem with zero user input, and no one takes issue with that. History and experience suggest that deciding which temporary is assigned to which register or stack slot is not a meaningful decision for a programmer. By contrast, decades of research have been invested into automatic parallelization of general-purpose code, like arbitrary C for-loops. However, identifying and expressing parallelism in an application seems more fundamental/important for the programmer to decide than register allocation. Automatic techniques have been ultimately eclipsed by DSLs with understandable, user-programmable models of parallelism like Halide [12] and Spark [16]. After all, autovectorization is not a programming model [10]. So how can program synthesis avoid a similar fate?

4 A Vision for Human-Centric Program Synthesis

The moral of these stories is that understanding the context, challenges, and capabilities of the working programmer is essential for improving the programming experience. Applying such a human-centric lens in designing and evaluating synthesis tools could accelerate the progress of synthesis research and promote the real-world adoption of these techniques. While the goal of this paper is primarily to spark discussion—what do you think human-centric synthesis entails?—I want to set the stage by articulating my principles for improving synthesis tools.

1. **Synthesis tools should use a user's most productive specification language.** Input/output pairs have been a popular specification language for synthesis, since "PBD is a natural match for artificial intelligence... by observing the actions taken by the user (training examples), the system can create a program (learned model) that is able to automate the same task in the future." [8] Moreover, for end-users without training in formal languages, I/O pairs are the highest level of abstraction at which they can formally

specify behavior. However, programmers can use a diverse array of representations for specifications. These range from testing (e.g. unit testing, randomized test generation [2]) to declarative languages (PlusCal, UML) to programming languages (sketches [13]).
Synthesis tools should use a specification language based on the difficulty of writing abstract rules versus writing individual examples in a given domain. Dana's window query may be easier for her to specify in Python, while a data structure manipulation like rotating a tensor may be easier to specify by examples. Human-centric evaluations of synthesis should seek to empirically characterize this trade-off.

2. **The synthesized program can not be a black box.**
Synthesis tools have historically been used like compilers: input the specification, and don't look at the output program, just run it. Again, while this approach works for end-users who may lack the technical knowledge to understand the synthesized program, such an interaction mode is rarely desirable for a programmer. Programs are written, re-read, tweaked, maintained, handed off to other collaborators, and so on. Professional programmers spend only 5% of their time writing code [9, 15].
Subsequently, Programmers must be able to comprehend and maintain synthesized programs. A synthesis tool should generate readable code and be able to explain its decisions, like Dana's imagined UI. Readability metrics can be informed by existing principles of programming notation design, like the cognitive dimensions framework [4].

These principles have helped me envision application spaces beyond the traditional purview of synthesis, like those characterized in the stories above. For example:

1. **Helping programmers learn new languages and APIs.** Programmers, whether hobbyists or full-time developers, encounter learning opportunities every time they code. Dana's was intentional: she realized she didn't know a feature and searched for it. But many more opportunities are passed by due to lack of awareness of a language or API feature. Anecdotally, I know Rust users that say the linter (Clippy) has helped them learn APIs through simple syntactic patterns. A synthesis tool as an extremely powerful linter could identify when someone likely doesn't know a concept ("there are 10 places in your code base that could be simplified with a for-loop"), highlight relevant code, and even suggest the translation if possible. By explaining its code-generating decisions, a synthesis tool can move beyond code that just works, to code that teaches how it works.

2. **Evaluating the impact of an API/language change.** When maintainers of libraries and languages debate new features, questions arise like: how many people would use this change? Would their code be meaningfully improved with this feature? For example, the Python community recently accepted the contentious PEP 572 "walrus operator." Guido was ultimately convinced by maintainers who combed through their own codebase, demonstrating dozens of places where the proposed feature could be applied [3]. A synthesis tool could help maintainers automate such exploratory tasks and more freely experiment with proposed designs.

Enabling these applications raises a number of exciting research questions in the design, implementation, and evaluation of synthesis tools. If high-level specifications replace I/O pairs, does this reduce the program search space, or is it just a means of generating examples (like QuickCheck)? Can API or language designers make their systems more amenable to synthesis? I hope that perspectives from the PL/HCI community can contribute greatly to these endeavors.

5 Acknowledgments

I would like to thank my advisor Pat Hanrahan for his everlasting support despite my constantly evolving research direction. And a big thanks to Brian Hempel, Georgia Gabriela Sampaio, and my anonymous reviewers for constructive comments that substantially improved the quality of this paper.

References

1 Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1205–1220. ACM, 2018.

2 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

3 Jake Edge. The pep 572 endgame. `https://lwn.net/Articles/759558/`, 2018.

4 Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

5 Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.

6 Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy java software to default methods. In *Proceedings of the 39th International Conference on Software Engineering*, pages 82–93. IEEE Press, 2017.

7 Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of java 8 streams. In *Proceedings of the 41st International Conference on Software Engineering*, pages 619–630. IEEE Press, 2019.

8 Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–65, 2009.

9 Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE Press, 2015.

10 Matt Pharr. The story of ispc. `https://pharr.org/matt/blog/2018/04/18/ispc-origins.html`, 2018.

11 Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 107–126. ACM, 2015.

12 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices*, volume 48, pages 519–530. ACM, 2013.

13 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

14 Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300. ACM, 2019.

15 Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

16 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.