Program Recognition in Synthesis

Michael B. James, Nadia Polikarpova

Abstract

Program synthesizers can offer a user many candidate snippets to fit a specification. How should a user decide which snippet is the right one for them? We introduce the problem of *program recognition* in the context of program synthesis. We argue recognition tasks are distinct from program comprehension, unique to synthesis, and under-explored. We run an exploratory study on program recognition and share our findings.

1 Introduction

Program synthesizers are now fast enough to suggest several code snippets at once. Such synthesizers now present a new problem to their users, they need to be able to recognize which snippet best matches their specification, if there even is one. But how should synthesizers and their related tooling aid a user in recognizing the snippet they actually want?

We call this setting *program recognition*. It is the general task of identifying a program matching some specification. Programmers have likely previously encountered a recognition task when searching Stackoverflow to determine if someone, somewhere on the internet has had the same problem and also posted a solution. This task has become uniquely important to modern program synthesizers whose selling points include being faster or safer than a human writing the snippet alone. To add to their value, synthesizers need to consider this last-mile of synthesis so both inputting the intent and choosing the right right snippet is easier than doing the programming task without the synthesizer. We argue program recognition is distinct from program comprehension, uniquely important to synthesis, and under-explored.

Program recognition in a synthesis context leaves many questions to answer. When do users reach for documentation, examples, or another tool to select a snippet? When users do reach for examples and how do they use or generate them? Do programmers compare against other snippets, and how? How much do users try to actually understand their chosen snippet? Can we use existing tools from program comprehension to aid in recognition?

Prior work has touched on principles of program recognition, but none have made it their principle concern. The H+ tool for Haskell shows several snippets and each snippet has a handful of differentiating examples [3]. Zhang et al. allow a user to request examples to test corner cases in regular expression synthesis [9]. To our knowledge, prior work assumes that input-output examples are the best way to choose a result from a synthesizer, but this assumption has not been empirically validated, nor this design space explored.

Contributions. The gap in the program synthesis literature is a question unasked and unanswered: how does a users choose the most appropriate code snippet from multiple synthesis results? In this paper we present the following:

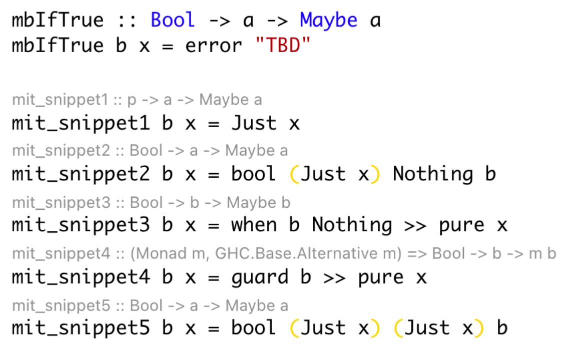
- We introduce the problem of *program recognition* in synthesis.
- We pose research questions in this topic.
- We perform an exploratory study, observing what techniques programmers deploy to choose a snippet from a synthesizer.

2 Motivating Example

Kelly is a functional programmer trying to write a small function in Haskell. Her function `mbIfTrue` should return an optional value from the input based on the boolean argument. If it is true, then return should exist otherwise it should be the empty-value. The function should have the type $\forall\alpha.$`Bool` $\rightarrow \alpha \rightarrow$ `Maybe` α. Kelly asks her IDE's synthesizer to complete the snippet given the type and it provides 5 possible results, as mocked-up in Figure 1. Kelly wants to quickly figure out if any of the snippets accomplish her task or see that none do, to refine her specification (or write the function herself). Current program synthesizers do not yet provide user support for this last-mile of program synthesis: choosing the right snippet.

We envision a synthesizer augmented with tools to help a user like Kelly recognize the snippet she needs. A synthesizer aware of common difficulties reading and choosing snippets might provide

PLATEAU
12th Annual Workshop at the Intersection of PL and HCI



```
mbIfTrue :: Bool -> a -> Maybe a
mbIfTrue b x = error "TBD"

mit_snippet1 :: p -> a -> Maybe a
mit_snippet1 b x = Just x
mit_snippet2 :: Bool -> a -> Maybe a
mit_snippet2 b x = bool (Just x) Nothing b
mit_snippet3 :: Bool -> b -> Maybe b
mit_snippet3 b x = when b Nothing >> pure x
mit_snippet4 :: (Monad m, GHC.Base.Alternative m) => Bool -> b -> m b
mit_snippet4 b x = guard b >> pure x
mit_snippet5 :: Bool -> a -> Maybe a
mit_snippet5 b x = bool (Just x) (Just x) b
```

Figure 1. The description on this task was, "Function test takes two inputs, a boolean value b and an arbitrary value x. It returns an optional value, which contains x if and only if b is True". It included one test case, not shown. Modern Haskell IDEs infer top level types and show them in light grey. Participants were asked to select all appropriate snippets. Only snippet 4 matches this specification.

information modern IDEs already surface like documentation and type information. A recognition-aware synthesizer might also show some generated input-output tests; these tests could show off different and useful properties of the snippets. Such a tool could allow easy simultaneous execution of the snippets to compare outputs. Some users might care about seeing intermediate values or intermediate type applications and their tool should support them. Without better understanding the process of recognition, we cannot meaningfully improve the snippet selection process in synthesizers.

3 Related Work

Program comprehension typically focuses on how a programmer understands an entire software system or part of it [5]. Comprehension research has focused on how a programming paradigm affects understanding [7]; how developers understand patches [8]; and more. Recognition differs from comprehension: in recognition, a programmer's goal might not be complete understanding, but simply to choose something that works *well-enough*—or to better understand their task.

Synthesizers that produce more than one result have vaguely pointed to this program recognition problem. For example, H+, a type-directed synthesizer, can produce many snippets and examples with each. The tool focused primarily about marrying types and/or examples as input, and secondarily on example generation–without necessarily asking what kind of examples a user would want to see [3]. Peleg and Polikarpova's BESTER synthesis engine provides multiple snippets and shows the user which examples a given snippet successfully passes [6]. This engine helps bring the community closer towards a tighter human-synthesizer interaction loop, but did not address how or why we might help users recognize the snippet they want when it's under their nose. WREX presents a data scientist readable Python that accomplishes a programming-by-example synthesis task [1]. WREX provides feedback, helping a user see if the synthesized code accomplishes their data-wrangling task; however, it is limited to Flashfill-like domains [2]. In more the tightly controlled domain of regular expressions, Zhang et al. help a user select a synthesized regular expression matching their intent [9]. A user can request more examples of a preferred style to aid in recognition.

All prior works assume that *examples* are the most efficient way to help a user with a program recognition task. To develop better synthesizers with a holistic approach to program recognition, we must validate or broaden this assumption.

4 Study Design

To determine how users recognize the snippet that matches their intent we ran an exploratory lab study. We recruited 4 participants to observe while they solved program recognition tasks. In this think-aloud study, each participant was presented with a small function to complete using any of the five program snippets provided to them. We wanted to get insight into what a user's process is to determine the best snippets for the given task.

Participants. We recruited participants with moderate experience with Haskell through a recent

Name	Type
mbIfTrue	`Bool → a → Maybe a`
firstJust	`a → [Maybe a] → Maybe a`
inverseMap	`[a → b] → a → [b]`
dedup	`(Eq a) => [a] → [a]`
applyNTimes	`(a → a) → a → Int → a`

Figure 2. The functions participants were asked to implement using snippets provided to them.

conference on functional programming and a recruitment form posted to Twitter. We recruited graduate students from two different institutions (P2, P3, P4) and one professional (P1). Both P1 and P3 have 10 years of experience with Haskell, while P2 and P4 have 2 and 5 years of experience, respectively.

Research Question. This need-finding study seeks to guide future work for synthesis tooling in program recognition tasks and we had one primary question: **What techniques do programmers use to recognize the best snippet in a situation?**

4.1 Setup

We provided each participant with a small Haskell repository so they could use their preferred tooling on their own machine. Only P1 had tight Haskell integrations into their IDE while P2, P3, and P4 needed to rely on Hoogle [1] and their REPL for types and documentation. Participants were told they could use any resource they like to determine which snippets worked best, including those on the web.

4.2 Tasks

Participants were given 5 tasks to complete. Each task contained a simple function to implement using the program snippets provided. Each function had 5 possible snippets, with at least one correct snippet per task. A task included an English-language description and one test case. The test case was designed to be unhelpful, often accepting all or most snippets without modification. Participants were told that there were zero or more possible solutions and they were asked to identify all snippets matching the description.

The tasks stress different aspects of functional programming, especially in how users would have to think about which snippets are appropriate (all shown in Figure 2). `mbIfTrue` requires reasoning about monadic behavior, common in elegant Haskell code. `firstJust` relied on oft-unused parts of the `Data.Maybe` module in the standard library, stressing how a user reasons through unfamiliar components. `inverseMap` and `applyNTimes` forced a user to reason through higher-order code snippets. `dedup`'s type is the least descriptive of the tasks: the function's type gives little insight into how the function must work unlike the others.

5 Observations

We observed our participants and gathered commonly used techniques.

5.1 Strategies

Each participant had a slightly different way of discovering the appropriate snippet for the task. Although every participant used a process of elimination, either saying aloud that a snippet was bad, commenting them out, or leaving a comment next to an eliminated candidate. P1, P2, and P3 each used a multi-pass approach: after reading the specification, they went through the snippets in order to reason through its type or its code. If a snippet was particularly challenging to understand, a participant would make a guess, note it, and move on to the other snippets. On one such task, P3 said, "don't love this monadic stuff" before adding a comment "ew" to a highly polymorphic snippet and moving on. Only after eliminating several snippets and presented with a choice would

1 A popular API search engine for Haskell. hoogle.haskell.org

these participants more deeply inspect the remainder. Each participant appeared to go through the snippets a different number of times: P4 went through the snippets a single time per task, but in great detail; while P1 would go through three times. The passes did not always cover the same things. P1 looked closely at the snippet types, P3 looked at the provided test both to find snippets that clearly wrong and can be eliminated.

Examples. Use of examples varied wildly. In task 4, P2 relied on only documentation to determine the correct snippet. Only after they declared their choice did they run the provided test (which would have accepted any snippet). Other participants used examples heavily, often in lieu of documentation. For example, P3 in tasks 1, 4, and 5, would look at the documentation for a component, then run that component with their own input. P3 described the documentation as being too long and that running the component was easier for them. This participant built his understanding of subexpressions by running them.

All participants appeared to have some transition point when a snippet became too complex to think about symbolically or with types, and had to think using examples. Two participants at some point just ran all snippets for a task through the same example, pointing to the general complexity of the snippets (task 1 and task 3).

P1 and P2 tended to run a test on all snippets under consideration at the same time. In other tasks, P1 and P3 kept only one example and slowly changed it as they eliminated snippet after snippet. The particular values used in the examples were never of particular interest beyond being distinct (except where duplication was part of the spec, in task 4).

Types. Haskell programming is synonymous type-directed programming but the types were a double-edged sword. Task 4 operates on lists and uses a equality typeclass constraint yet no participant considered the typeclass. On the other hand, P1, P3, and P4 were confused by the name of task 3, `inverseMap`, but after considering its type, they each explained having an insight into how they would expect the right snippet to work.

The highly polymorphic nature of Haskell comes with a mental cost. Every participant was confused by a snippet using the identity function in place of function application in a higher order function. Even an expert (P3) thought the snippet was ill-typed. Participants struggled to understand how this snippet could typecheck. We believe that, like intermediate *values*, intermediate *types* and *type applications* could help recognition tasks with higher-order functions.

P1's use of types was exceptional. This participant looked at snippets and the specification to determine if they were relevantly typed (*i.e.* each argument must be used at least once), and was able to eliminate errant snippets in task 1. P1 was the only participant whose IDE presented all inferred and un-annotated types, making it easier to use this extra information.

Fixing snippets. Participants were inclined to modify snippets in primarily two different ways. In the first way, some participants wanted to de-sugar partially applied functions into eta-long form (P2, P4). This behavior only came up in higher-order snippets. The second way was to fix an incorrect snippet to fit the specification. P1, P3, and P4 all suggested fixes for snippets to make them correct, yet none of them claimed they would have come up with the same snippet on their own.

5.2 Takeaways

All users in our study used some form of process of elimination, typically by marking some candidates as out of consideration. Users would likely benefit from some way to maintain that state while reading through synthesized snippets. We believe that even in a functional setting like in Haskell, examples are still useful in program recognition. Our study confirms observations from Glassman's work that it's easier for users to modify things than to invent new things [9], but in our case this applies both to programs and to examples. Users seem to benefit from local information, for subexpressions, both at the level of values and the level of types. Lastly, since several participants saw ways to change snippets to fit the specification, synthesizers may wish to embrace this interaction mode. Such an interaction model may allow a program synthesizer to act more as *program exploration* tool, which will be especially useful in a neurally-guided setting where the synthesizer may not possess a semantic understanding, but can nonetheless guide a user to solving their task.

References

[1] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A unifed programming-by-example interaction for synthesizing readable code for data scientists," p. 12, 2020.

[2] S. 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*, ser. POPL 11, Association for Computing Machinery, Jan. 2011, pp. 317–330, ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. [Online]. Available: https://doi.org/10.1145/1926385.1926423.

[3] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova, "Digging for fold: Synthesis-aided api discovery for haskell," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 205:1–205:27, Nov. 2020. DOI: 10.1145/3428273.

[4] S. Lau, S. S. Srinivasa Ragavan, K. Milne, T. Barik, and A. Sarkar, "Tweakit: Supporting end-user programmers who transmogrify code," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, May 2021, pp. 1–12, ISBN: 978-1-4503-8096-6. [Online]. Available: https://doi.org/10.1145/3411764.3445265.

[5] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, 31:1–31:37, Sep. 2014, ISSN: 1049-331X. DOI: 10.1145/2622669.

[6] H. Peleg and N. Polikarpova, "Perfect is the enemy of good: Best-effort program synthesis," p. 30, 2020.

[7] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1125–1143, Dec. 2017, ISSN: 1939-3520. DOI: 10.1109/TSE.2017.2655524.

[8] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE 12, Association for Computing Machinery, Nov. 2012, pp. 1–11, ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393656. [Online]. Available: https://doi.org/10.1145/2393596.2393656.

[9] 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*, ACM, Oct. 2020, pp. 627–648, ISBN: 978-1-4503-7514-6. DOI: 10.1145/3379337.3415900. [Online]. Available: https://dl.acm.org/doi/10.1145/3379337.3415900.