UNIVERSITY OF CALIFORNIA SAN DIEGO

Exploratory Phenomena in Program Synthesis

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

in

Computer Science

by

Michael Buchanan James

Committee in charge:

Professor Nadia Polikarpova, Chair
Professor Michael Coblenz
Professor Philip Guo
Professor Sorin Lerner

2024

Copyright
Michael Buchanan James, 2024
All rights reserved.

The dissertation of Michael Buchanan James is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

I could not possibly thank all the people and places that have made this degree possible for me in the space I have. I dedicate this dissertation to all those who had a significant impact on my decision to attend and complete graduate school.

I dedicate this to my family. To Frank, and his comedy through the years. To Tom, for his elder wisdom. To Dad, the most wholesome and genuine person I know. To Mom, who always encouraged me to stay true to myself and rooted for me when I needed it the most.

Carolyn, Dustin, Evan, Tony: I dedicate this to you who encouraged me, in word, in spirit, or in being to apply and attend graduate school.

To my friends who have made San Diego a special place. Matt Kolosick, he challenged me to better myself and think independently. Nikolai Vogler, his coffee and company will always be welcome chaos. Evan Johnson, whose conversations are always top quality. David Thien, the best story teller I know. John Renner, he helped me choose UCSD, and continued to make it an excellent place. Daniel Spokony, for his adventuring and advice.

To my gaggle who showed me what queer community looks like. Mac Lockhart, I have missed his athleticism-fueled friendship since graduation. Connor Redpath, whose absurd stories always spark joy. Max Silva, and his RPDR dedication. William Howard, I will miss his push for adventures. Shane Xuan, for showing me how to game any system.

Thanks to my friends not in San Diego, who supported me from afar. Carolyn Saund, she is a constant inspiration of brilliance, wit, and determination. Rich Wenner, I always looked forward to our catch up sessions and to riffing off each other. Paul Templeton, I'm thankful for his warmth and eternal friendship: whenever we see each other it's like no time has passed. Thank you Javier, for his care, deadline discipline, and for teaching me to be a global citizen over those three years. I look back fondly on those adventures.

Lastly a thank you to my loving partner Jason whose compassion, grace, and curiosity inspire me. Your consistent excitement and passion is brilliantly infectious.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Nadia Polikarpova. I would not have completed this degree if not for her guidance, patience, insight, and enthusiasm. From the first research meeting I had with Nadia, I knew that researching with her would be an exciting journey. She is a phenomenal, efficient, and engaging author, and I'm thankful for how she has shaped my writing. Through all of it, I am immensely thankful to Nadia.

Thank you to Elena Glassman, whose consistent optimism and enthusiasm has helped keep a project going, and her countless ideas have helped progress projects.

Thank you Evan Czaplicki. His vision demonstrated that programming languages research is itself a kind of human-computer interactions research.

Chapter 1, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA. Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Nadia Polikarpova. ACM 2020. The author was a principal author and investigator on this work.

Chapter 2, in part, is a reprint of the material as it appears in the 12th Annual Workshop at the Intersection of PL and HCI. Michael B. James and Nadia Polikarpova. PLATEAU 2021. The author was a principal author and investigator on this work.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 7, Issue OOPSLA1. Shraddha Barke, Michael B. James, Nadia Polikarpova. ACM 2023. The author was a principal author and investigator on this work.

Chapter 4, in part, is a reprint of the material as it will appear in the Proceedings of the 2024 CHI Conference on Human Factors and Computer Systems. Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, Sorin Lerner. ACM 2024. The author was a principal author and investigator on this work.

The conclusion, in part, is a reprint of the material, current being prepared for submission

for publication, with the following coauthors: Michael B. James, Emmanuel Anaya Gonzalez, Mark Barbone, Elena L. Glassman, Nadia Polikarpova. The author was a principal investigator on this work.

VITA

2015	B. S. in Computer Science, Tufts University
2015-2016	Software Engineer, Uber Technologies
2017-2018	Software Engineer, Jana Mobile
2021	M. S. in Computer Science, University of California San Diego
2024	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, Sorin Lerner. "Validating AI-Generated Code with Live Programming". CHI Conference on Human Factors in Computing Systems 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.

Michael B. James, Nadia Polikarpova. "Program Recognition in Synthesis." 12th Annual Workshop at the Intersection of PL and HCI. Plateau Workshop 2022.

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, Nadia Polikarpova. "Digging for Fold: Synthesis-Aided API Discovery for Haskell." Proceedings of the ACM on Programming Languages. Volume 4 (OOPSLA). November 2020.

Zheng Guo, Michael B. James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, Nadia Polikarpova. "Program Synthesis by Type-Guided Abstraction Refinement". Proceedings of the ACM on Programming Languages. Volume 4 (POPL), Article 12. January 2020.

ABSTRACT OF THE DISSERTATION

Exploratory Phenomena in Program Synthesis

by

Michael Buchanan James

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Nadia Polikarpova, Chair

Program synthesizers promise to save time for the programmer by writing parts of their code for them, provided the programmer knows exactly what they want. For it to suggest that code, the programmer must convey the nuances of their specification to the synthesizer. Is there a place for program synthesizers when those nuances are unclear, and a programmer must explore the design space of their problem? This dissertation argues that program synthesizers can facilitate exploring for code, and that techniques to validate explored code are needed to reduce cognitive effort.

To use a traditional synthesizer successfully, the programmer needs a deep understanding of the problem. Firstly, a programmer must convert their intent into the specific language

their tool understands, which may not match how they think about their intent. Secondly, a programmer needs to know all the edge cases of their problem and how each one should behave. If both of these cannot be met, then a traditional synthesizer will usually be more effort to use than simply writing the desired program.

As modern, probabilistic synthesizers require less overall information to offer a suggestion, they can fill a new role in the programming workflow: code exploration. A programmer no longer needs complete understanding of their problem nor be able to communicate all nuances to the tool. This new exploration domain presents new challenges, particularly one of *validating* the code matches the often incomplete intent of the programmer. Validating code against intent must be quicker or easier to justify a tool's use in exploration to be useful. In this dissertation, I show that both traditional and probabilistic program synthesizers can aid in this exploratory process of programming, and that techniques to validate that explored code reduce the cognitive effort.

Introduction

Program synthesizers promise to save time for a programmer by writing parts of their code for them. Presently, these tools work best when a programmer knows exactly what they want. They must be able to convey the nuances of their specification to their tool, but these tools typically only accept one type of input [46, 97, 2, 6, 44, 83]. Can program synthesizers still offer assistance to a programmer when that specification is not clear?

Some times that specification cannot be conveyed to a program synthesizer because the specification is too complex. These specification styles are highly specialized (*e.g.* logical constraints [44, 95, 109], specialized types [94, 53, 62], carefully constructed examples [1, 75, 84, 37, 71]), requiring an expert user to correctly formulate. Each has their own set of tradeoffs, between expressiveness and simplicity. So, even if a programmer knows exactly what they want to generate, there may be difficulty in expressing their intent in just one style.

Other times, a programmer does not have a clear picture of what exactly they need at all. Synthesis is a search problem: but, without a destination pinpoint, classical synthesizers struggle to offer useful suggestions. Insufficient specification often leads to long lists of trivial programs. The situation with classical, search-based synthesizers make them well suited for problems like optimization [92, 105].

Modern, probabilistic synthesizers require less overall information to offer a suggestion. They operate simply off the past words typed, whether they are structured or not [19]. Those past words do not need to form a complete specification, or even a complete thought. Such a freeform

input invites these model-based synthesizers into a new role in the programming workflow: code exploration.

While exploring for code, a programmer no longer needs an a priori understanding of their problem. Instead, they can offer what they know, and look for pieces of information that may lead them to usable code. While exploring for code with the aid of a synthesizer, the programmer must then consider one or more new pieces of code to determine whether or not it fits their intent. This presents a new challenge, particularly of *validating* that the code matches the often incomplete intent of the programmer. Validating the code from a synthesizer requires some cognitive effort.

This exploratory phase of programming is not often associated with program synthesis, yet, there is still a place for synthesizers in the workflow. To fit in a workflow, exploratory program synthesizers will require affordances to easily navigate the code they generate. This dissertation argues that program synthesizers can facilitate exploring for code, and that techniques to validate explored code are needed to reduce cognitive effort.

Overview

Chapter 2: Human-Centric, Type-Directed Program Synthesis

This chapter presents a multi-modal program synthesizer for Haskell. This tool empowers a programmer to use types, examples, or both to compose highly polymorphic library calls. Combining these different search inputs allows a user to create their specification more ways that make sense to them, helping to span the gulf of execution. This tool provides additional information on the generated calls to help a user understand and select the appropriate synthesized code. We evaluated the utility of this synthesizer in comparison to a common Haskell developer tool, marking the first user-study of a synthesizer. We find that programmers complete their tasks faster and are able to solve more tasks in the given time with the aid of this tool. The study

solidifies that synthesizers are ready for broader usage, as they can compete with off-the-shelf non-synthesizer tools.

Chapter 3: Program Recognition in Synthesis

This chapter identifies a core problem in the usability of program synthesizers. While program synthesizers can offer many candidate snippets to fit a specification, how should a user decide which snippet is correct for their use case? This chapter introduces the problem of "program recognition" in the context of program synthesis, and argues it is distinct from the well-researched topic of "Program Comprehension". Reframing the needs of a programmer from those of comprehension to those of mere recognition enables a more tailored experience in program synthesis. Recognizing a program is equivalent to overcoming the gulf of evaluation. Synthesizers can focus on providing enough detail to allow a user to find the correct program for their needs, without necessarily deeply understanding the produced code.

Chapter 4: A Theory of Code-Generating Model Usage

This chapter explores how programmers interact with AI-powered program synthesizers. This chapter presents a grounded theory analysis of how programmers use the Copilot[38] programming assistant. By observing 20 participants, with a range of prior experience using the assistant, we produce a *bimodal* theory of interactions. Programmers are either in an *acceleration* mode where the tool is helping them put thought to code quickly; or, they are in *exploration* mode wherein a programmer is unsure how to proceed and uses Copilot to explore options. The theory we generate provides a range of possible opportunities for improving future AI assistants.

Chapter 5: Validating AI-generated programs with live programming

Throughout the findings in chapter 3, we find that programmers wanted to *validate* suggestions from the assistant in their own case. In this chapter, my coauthors and I combine an AI assistant with projection box [66] from live-programming. We show that we can reduce cognitive burden with this technique combining both live-programming and large-language models. This chapter demonstrates that validation techniques on explored code reduce cognitive effort.

Conclusion: Future Work

The conclusion outlines future work to assist programmers explore and validate suggestions from a large-language model. chapter 3 identified that programmers spend significant time exploring the design space of their problem with the aid of an AI-assistant. The cost of inferring programs is currently high but there is much work to lower that cost [103]. With more samples, we can present a programmer with a better understanding of the space of possibilities, instead of just one concrete suggestion. We can use existing program analysis techniques to group programs and present commonalities.

Chapter 1

Human-Centric, Type-Directed Program Synthesis

1.1 Introduction

Consider the task of implementing a function `dedup` that eliminates adjacent duplicate elements from a list (*e.g.* `dedup [1,1,2,2,1] = [1,2,1]`). In a functional language like Haskell, this task can be accomplished without explicit recursion, simply by using functions from the standard library:

```
dedup xs = map head (group xs)
```

This solution first calls `group` on the input list to split it into clusters of adjacent equal elements (*e.g.* `group [1,1,2,2,1] = [[1,1], [2,2], [1]]`), and then `map`s over the result to extract the `head` of each cluster. This implementation is not only shorter than a recursive one, but also more idiomatic. But how is the programmer to *discover* this solution?

The need for such discovery is particularly acute in functional languages, whose expressive types and higher-order functions make libraries extremely versatile and compositional.

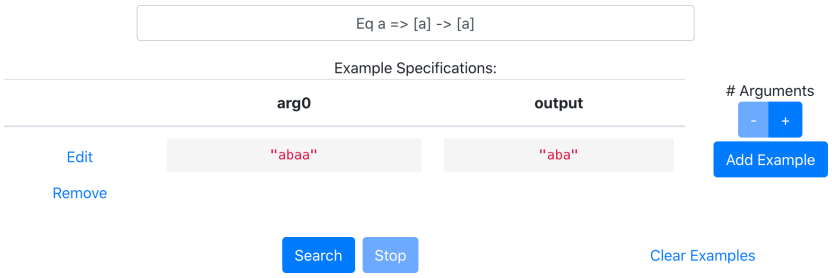


Figure 1.1: In HOOGLE+, the user can search for `dedup` using its type, one or more tests, or both.

As a result, discovery is especially *useful* as many computations can be expressed by gluing components from existing libraries. At the same time, discovery is especially *difficult* as library functions are very general and can be composed in myriad ways. Online help forums like STACKOVERFLOW only contain solutions for common programming tasks, and are generally less helpful outside of a handful of most popular programming languages. As an alternative, Haskell programmers often turn to the HOOGLE API search engine [77] to search for library functions by their type; but HOOGLE only helps if there is a single library function that does the job, which is not the case for `dedup` where we must compose multiple functions into a *snippet*. Our goal is to bridge this gap and build an API discovery tool for Haskell that helps programmers find snippets like our implementation of `dedup`.

Type-Directed Component-Based Synthesis. The core technical challenge for API discovery is how to efficiently search the space of all snippets when the API library has hundreds or thousands of functions. *Component-based program synthesis* techniques [73, 47, 33, 46] tackle this challenge using a type-directed approach. In particular, our prior work on synthesis by *type-guided abstraction refinement* (TYGAR) [46] demonstrates how to efficiently perform type-directed search in the presence of polymorphism and higher-order functions, which are ubiquitous in functional languages. In this work we build upon the TYGAR search algorithm to implement an API discovery tool we dub HOOGLE+.

Challenges. Although the core search algorithm behind HOOGLE+ is not new, turning

this algorithm into into a practical API discovery tool required overcoming three important challenges.

1. **Specification:** The first challenge is that of specification: how should the programmer communicate their intent to the synthesizer? In Haskell, *types* are a powerful and concise way to specify program behavior thanks to parametric polymorphism, which significantly restricts the space of possible implementations of a given type. Types are the preferred mode of specification for HOOGLE users and moreover, TYGAR requires a type in order to perform snippet search. The flip side of expressive types is that a Haskell beginner might not immediately know the most appropriate type for the function they want to implement. Consider `dedup`: its most general type is `Eq a => [a] → [a]`; this type is polymorphic in the list element, but restricts these elements to be *equatable*, because `dedup` has to compare them for equality. When types become non-trivial, it is more natural for a user to specify their intent using *input-output tests*. Based on these observations, we design HOOGLE+ to allow three different modes of intent specification: *only types*, *only tests*, or *both* (see Figure 1.1). To enable type-directed search when the user only provides tests, we develop an algorithm to *infer* types from the tests. Note that there might be many types of different levels of generality that are consistent with the tests, so HOOGLE+ presents a set of *likely type specifications* to the user, as shown in Figure 1.2.

2. **Elimination:** Specifications are often ambiguous, especially when the user provides the type signature alone. In this case TYGAR might return many irrelevant candidate programs. For example, searching for `dedup` by its type might generate programs like `\xs → []` (which always returns the empty list) or `\xs → head []` (which always crashes by taking the head of an empty list). Intuitively, these programs are clearly uninteresting, and we shouldn't need additional user input to eliminate them from the synthesis results. To address this challenge, we have developed an efficient heuristic for identifying uninteresting candidates using *property-based testing* [23, 101].

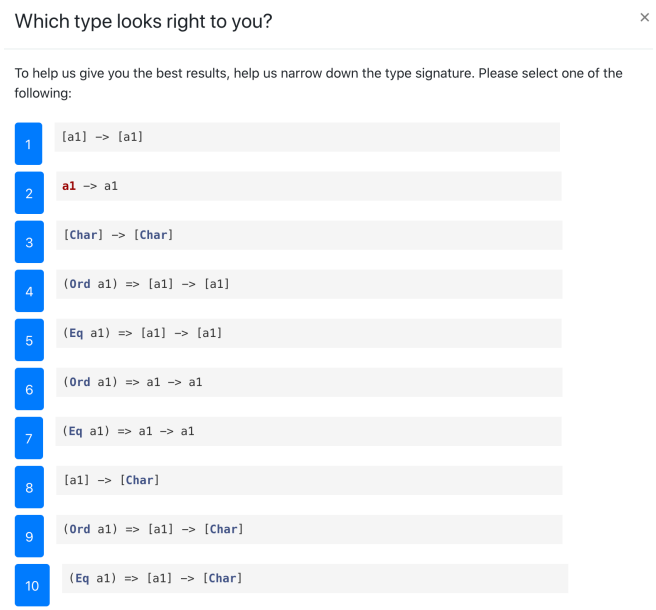


Figure 1.2: Candidate types for `dedup` inferred from the test `"abaa"` → `"aba"`.

3. **Comprehension:** Finally, once the candidate programs have been generated: how should the programmer decide which, if any, synthesis result solves their problem? To facilitate comprehension of a candidate program, HOOGLE+ automatically generates several examples of its behavior as shown in Figure 1.3. Unfortunately, a naive exhaustive or random generation yields many uninformative examples. We show how to address this challenge by relying, once again, on property-based testing to generate inputs with certain desirable qualities, such as examples of success and failure and examples that differentiate this candidate from the rest.

HOOGLE+. We have incorporated the three techniques described above together with the TYGAR search algorithm into a web-based API discovery engine. Figure 1.1 illustrates using HOOGLE+ for our running example: the programmer has specified the Haskell type signature for `dedup` and one example of its behavior. Figure 1.3 shows the list of candidate programs returned by HOOGLE+ (with the correct solution at the top).

User study. Does synthesis-aided API discovery actually help programmers solve their tasks compared to a more traditional workflow? We evaluate this question by conducting a

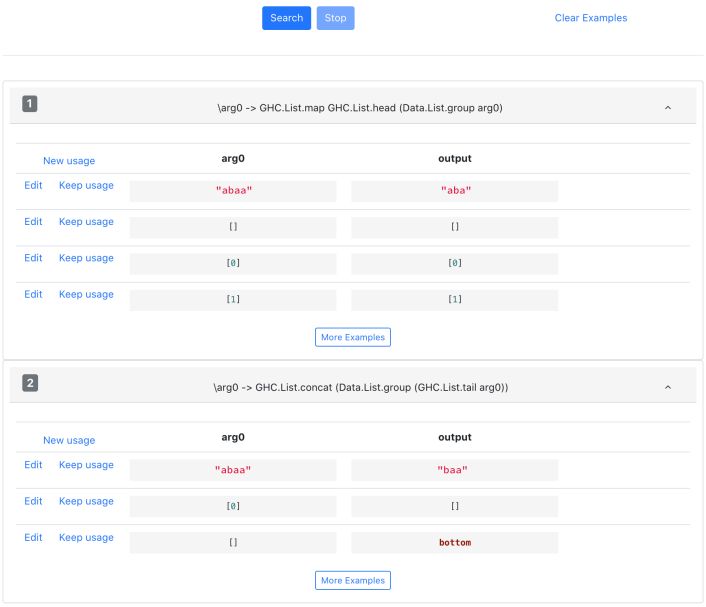


Figure 1.3: Two candidate solutions for `dedup`. The behavior of each solution is illustrated with both user-provided and auto-generated examples.

user study with 30 participants of varying levels of Haskell proficiency. The participants were asked to solve various programming tasks (including `dedup`) either using HOOGLE+ or using a popular code search workflow (HOOGLE together with an interpreter). The study shows that HOOGLE+ enables programmers to solve tasks faster and increases their success rate in finding a correct solution by more than 50%.

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

1. HOOGLE+, the first practical API discovery tool for a functional language with higher-order functions and polymorphic types; the tools accepts specifications in the form of types, input-output tests, or both, and displays candidate snippets together with examples of their behavior (section 1.2).

2. A new algorithm that infers likely type specifications from tests (section 1.4).

3. A new technique for automatically eliminating uninteresting synthesis results using property-based testing (see subsection 1.5.1).

4. A new technique for automatically generating examples of program behavior using property-based testing (see subsection 1.5.2).

5. The first user study evaluating the usefulness of synthesis-aided API discovery in a functional language (section 1.7).

1.2 Overview

We begin with an overview of the challenges to practical synthesis-aided API discovery and show how HOOGLE+ overcomes these challenges. We postpone the description of the core synthesis engine, type-guided abstraction refinement (TYGAR), to section 1.3.

1.2.1 Specification

Consider a user tasked with implementing our running example `dedup`, and assume that the user has a test in mind: on input `"abaa"` the snippet should return `"aba"`[1]. In order to make use of the TYGAR synthesis engine, however, the user also needs to provide a *type signature* for `dedup`, which the engine uses to efficiently navigate the search space.

Problem. Our user study shows that Haskell beginners often do not know the most appropriate type signature for the snippet that they are looking to implement (section 1.7). In particular, *typeclass* constraints are particularly tricky for beginners. For example, the appropriate type specification for `dedup` is `Eq a => [a] → [a]`, where the constraint `Eq a =>` allows using equality checks on `a` to remove the duplicates, but the need for this constraint is not obvious from the task description. Consequently, the user might search using the *overly general* type `[a] → [a]`, which will prove fruitless (the `dedup` snippet does not check against this type). Alternatively, the user might search an *overly specific* type, like `[Char] → [Char]`, which will yield too many results to be useful.

[1] In Haskell, a string is just a list of characters, so `dedup` can operate on strings.

Solution: Types from Tests. We address the specification problem with a novel technique that automatically *infers* likely type specifications from tests. In our running example, the user enters their intended test `"abaa"` → `"aba"` and leaves the type specification blank. HOOGLE+ then presents the user with a list of up to 10 candidate types, as shown in Figure 1.2. Notice that the correct type `Eq a => [a] → [a]` is listed in position 5. When reminded of typeclass constraints explicitly, users can often figure out which constraints they need.

Inferring likely type specifications is a difficult problem: as we show in subsection 1.6.1, there can be anywhere from a dozen to a *few million* types of different levels of generality consistent with a given set of tests. To pick a few likely candidates, our inference algorithm incorporates two new mechanisms: (1) a *filtering* mechanism eliminates candidate types that cannot be inhabited by a meaningful program (*e.g.* the type `[Char] → [a]` is eliminated, because any program of this type always returns the empty list) (2) a *ranking* mechanism that prioritizes simple and general types. We describe our inference algorithm in detail in section 1.4 and evaluate it empirically in subsection 1.6.1.

1.2.2 Elimination

Now consider a scenario where a user is searching for `dedup` by only its type, `Eq a => [a] → [a]`.

Problem: Meaningless and Duplicate Results. Type-only specifications are often highly ambiguous: although polymorphic type signatures help narrow down the search space, there might still be too many programs that check against a given type. Fortunately, many of these programs are clearly uninteresting, and can be *eliminated* without requiring additional input from the user.

We have identified two main sources of uninteresting synthesis results. *Meaningless* programs are those that crash or diverge on every input. For example, any program that contains the subexpression `head []` is meaningless as it *always* crashes re-

gardless of the input. The second source of uninteresting synthesis results are *duplicates*, *i.e.* semantically equivalent programs. For example, one candidate solution for `dedup` may be `\xs → init (head (group xs))` and the following one could be `\xs → tail (head (group xs))`. The two candidates syntactically differ in that they take the prefix (`init`) and suffix (`tail`) of the result of `head (group xs)`. However, they are semantically equivalent as the input to `init` and `tail` is always a non-empty list of identical values, *e.g.* `init ['a', 'a']` and `tail ['a', 'a']` are both equal to `['a']`. Our goal is to eliminate meaningless and duplicate programs from the output of HOOGLE+ automatically.

Solution: Property-Based Testing. In principle, the problem of determining whether a program is meaningful or whether two programs are equivalent is undecidable. In practice, however, it turns out to be sufficient to test these properties on a finite set of inputs. To do so efficiently, HOOGLE+ relies on the the SMALLCHECK property-based testing library [101].

Specifically, for each new candidate program p generated by the synthesis backend, HOOGLE+ invokes SMALLCHECK to test whether there exists some input where p produces an output within a given timeout. If this check succeeds, then for each previously displayed candidate p', HOOGLE+ asks SMALLCHECK to find a *distinguishing input* [56] for p and p', *i.e.* an input where they produce different outputs. For example, assume that HOOGLE+ has displayed several results for `dedup`, including the program $p' =$ `\xs → init (head (group xs))`, and the newly generated candidate p is `\xs → tail (head (group xs))`. This candidate passes the meaningfulness check (SMALLCHECK finds the input `[0]` where p returns `[]`), but fails the uniqueness check: after exhaustively searching all lists up to a given length and range of values, SMALLCHECK is unable to find an input where the output of p differs from the output of p'. Based on the failed uniqueness check, HOOGLE+ eliminates p from the list of results presented to the user.

Haskell's laziness presents a subtle challenge for test-based elimination: in Haskell, it

is common practice to write functions that produce infinite data structures, and such functions should be considered meaningful. At the same time, trying to print the output of such a function or compare two infinite outputs would lead to non-termination. HOOGLE+ builds upon the `ChasingBottoms` library [25] to ensure proper handling of infinite values.

1.2.3 Comprehension

HOOGLE+ displays a sequence of meaningful and unique candidates to the user, but how is the user to know *which* result implements their requirements? While some experienced programmers might be able to recall the behavior of the components well enough to mentally reconstruct the semantics of their composition, most users require further assistance to understand how each candidate behaves. One way forward is to show the user input-output *examples* for each candidate. However, there are two questions that must be addressed to facilitate example-based comprehension.

Problem: Comprehension Conflicts. First, *what* kind of examples is the user looking for? There is no "best" example for a candidate program as there are a range of different comprehension goals that a user might have for each candidate. They may try to *differentiate* that candidate from other similar snippets or they might be trying to understand the *functionality* of the candidate itself.

Solution: Multiple-Objective Witnesses. HOOGLE+ supports multiple comprehension objectives by generating input-output examples that serve to witness different properties of the candidate, namely: 1. Meaningfulness 2. Uniqueness 3. Functionality. In Figure 1.3, the first candidate's examples demonstrate these properties in order. The first example, with input-output (`"aab","ab"`), repeats the test specification. The next example, (`[],[]`) witnesses that this candidate is meaningful. The subsequent example (`[0], [0]`) serves a differentiating objective: the same input produces different output on candidate program #2. The last example, (`[1], [1]`), demonstrates the functionality of the candidate program, more such examples are

available on demand from the "More Examples" button. Note that in candidate program #2, the input `[]` demonstrates that the candidate is a partial function.

Problem: Minimality vs Interactivity. Second, *when* should examples be shown to the user? HOOGLE+ could wait until we have all candidates and *then* generate examples. On the plus side, waiting would let us find fewer inputs (or even just one) to differentiate each candidate. Unfortunately, the resulting lack of interactivity could drive users away from the tool altogether.

Solution: Laziness. Instead, we stream input-output examples with every candidate, providing more examples to already-displayed candidates, which may be hidden from view until the user clicks "More Examples" to avoid cluttering the UI. In the case of `dedup`, a user might see a *usage table* as seen in Figure 1.3. This table shows inputs along with their output for that candidate program. A user can edit the input for this usage and see the new corresponding output with the "edit" button on the left. A user can input their own usage with the "New Usage" button on the top left. Finally, a user can ask for more examples explicitly from the system with the "More Examples" button.

1.3 Background

HOOGLE+ builds upon prior work from two sources. First, the core program synthesis algorithm comes from our own prior work on *type-guided abstraction refinement* (TYGAR) [46]. That work developed a novel search technique but we did not focus on end-to-end usability of the synthesizer. Second, we filter candidate programs with the help of exhaustive testing framework SMALLCHECK [101].

1.3.1 Type-Guided Abstraction Refinement

In our prior work [46] we developed TYGAR, a component-based synthesis algorithm that takes as input a Haskell type and a set of library functions, and returns a list of programs of

the given type, composed from the library functions. Like prior work in component-based synthesis [73, 47, 33], TYGAR reduces the synthesis task to graph search; the challenge, however, is that in Haskell polymorphic components can infinitely explode the graph to search through. The key insight to overcome that explosion is to build a graph over *abstract types* which represent a potentially unbounded set of concrete types. We showed how to use graph reachability to search for candidate programs over those abstract types, and introduced a new algorithm that uses *proofs of untypeability* of ill-typed candidates to iteratively refine the abstraction until a well-typed result is found. TYGAR uses a *relevant* type system to ensure that every argument is used at least once in a candidate program.

Although TYGAR was able to produce a stream of well-typed candidates, our own experience during its empirical evaluation identified several shortcomings that had to be fixed in order to turn it into a practical API discovery tool. Firstly, for some type queries it returned too many uninteresting (meaningless or repetitive) programs. Secondly, it required the user to describe every programming task using its most general type, which can be challenging for beginners. Finally, it was often difficult to analyze synthesis results simply by looking at the generated code. We address these three shortcomings in present work.

1.3.2 SmallCheck

SMALLCHECK is a *property-based testing* framework for Haskell [101]. Property-based testing takes as input a *property*, *i.e.* a Boolean Haskell function with one or more arguments, and executes this function on a set of input values in an attempt to find a *counterexample*, *i.e.* an input where the property does not hold. While the original property-based testing framework QUICKCHECK [23] uses random input generation, its successor SMALLCHECK generates inputs exhaustively up to a user-provided *constructor depth*. As a result, SMALLCHECK always find the smallest counter-example to a property.

1.4 Type Inference from Tests

In this section, we detail our algorithm for inferring *likely type specifications* from tests. In a simply-typed language this is a straightforward task, since any function the user might want to synthesize has a unique, concrete type, which must coincide with the type of the test: for example, if the test is `"abaa"` $\rightarrow$ `"aba"`, the intended type specification must be `String` $\rightarrow$ `String`. In a language like Haskell, however, intended type specifications are often *polymorphic*, which poses two main challenges for type inference:

1. ***Reconciling multiple tests.*** Consider user input with two tests: `"abaa"` $\rightarrow$ `"aba"` and `[1,1,1]` $\rightarrow$ `[1]`, whose concrete types are [`Char`] $\rightarrow$ [`Char`] and [`Int`] $\rightarrow$ [`Int`], respectively. To reconcile these tests we must find a polymorphic type that can be instantiated into either of the two concrete types: for example, $[\alpha] \rightarrow [\alpha]$. To tackle this challenge, we build upon prior work on *anti-unification* [93, 99].

2. ***Generalizing from tests.*** Now consider user input with a single test `[1,1,1]` $\rightarrow$ `[1]`. The concrete type of this test is [`Int`] $\rightarrow$ [`Int`], but this behavior can also be produced by a function with a *more general* type, such as [`Int`] $\rightarrow [\alpha]$, $[\alpha] \rightarrow$ [`Int`], $[\alpha] \rightarrow [\alpha]$, or $\alpha \rightarrow \beta$. It is not obvious which one of these types would make the best specification: more general types reduce the search space and hence yield better synthesis results, but generalize too much and you will miss the intended solution. To tackle this challenge we propose a ranking heuristic to identify which generalized types are more likely to match user intent.

In section 1.4.1–section 1.4.5 we formalize our base algorithm for a simplified setting, where all tests have an unambiguous concrete type and the type system does not have type type classes. The base algorithm is extended to deal with ambiguous tests in section 1.4.6 and type classes in section 1.4.7.

1.4.1 Preliminaries

We formalize our base type inference algorithm for a core language defined in fig. 1.4.

Types. The language is equipped with a standard prenex-polymorphic type system: types T are either type variables, type constructor applications $C\ \overline{T}$[2], or function types. Type variables are denoted with lower-case Greek letters $\alpha, \beta, \ldots$. For lists, we use the familiar notation $[T]$ as syntactic sugar for `List` T. All type variables are implicitly universally quantified at the top level. A type T is *concrete* if is contains no type variables.

Type ordering. A *substitution* $\sigma = [\alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n]$ is a mapping from type variables to types that maps each α_i to T_i and is the identity mapping elsewhere. We write σT to denote the application of σ to type T, which is defined in a standard way. We say that type T is *more general* than type T' (or alternatively, that T' is *more specific* than T) written $T' \sqsubseteq T$, iff there exists σ such that $T' = \sigma T$. For example, $[\texttt{Int}] \sqsubseteq [\alpha] \sqsubseteq \beta$. The relation $\sqsubseteq$ is a partial order on types, and induces an equivalence relation $T_1 \equiv T_2 \triangleq T_1 \sqsubseteq T_2 \wedge T_2 \sqsubseteq T_1$ (equivalence up to variable renaming).

We say type T' is a *common generalization* of a set of types $\overline{T_i}$ if $\forall i. T_i \sqsubseteq T'$. The *least common generalization* (or *join*) of $\overline{T_i}$ always exists and is unique up to $\equiv$, so, by slight abuse of notation, we write it as a function $\sqcup(\overline{T_i})$. For example, $[\texttt{Char}]$ and $[\texttt{Int}]$ have two common generalizations, $[\alpha]$ and β, and $[\texttt{Char}] \sqcup [\texttt{Int}] = [\alpha]$, the more specific of the two.

Type checking. We omit the exact syntax of terms e, apart from the fact that they include values v. Tests t are built from argument values and a result value. We also omit the definition of typing environments Γ, which hold the types of data constructors and binders for $\lambda-$ terms, and term typing, since they are entirely standard; instead we assume access to a *type checking oracle* $\Gamma \vdash e :: T$, which decides whether term e checks against type T in Γ and a *type inference oracle* $\Gamma \vdash e \Longrightarrow T$, which computes the most general type T such that $\Gamma \vdash e :: T$ holds. Our implementation uses GHC to implement both oracles.

[2]Throughout this section, we write $\overline{X}$ to denote a sequence of syntactic elements X.

T	$::= \alpha \mid C\ \overline{T} \mid T \to T$	*Types*
σ	$::= [\overline{\alpha \mapsto T}]$	*Substitutions*
e	$::= v \mid \ldots$	*Terms*
t	$::= v \mid v \to t$	*Tests*
	$\Gamma \vdash e :: T$	*Type checking*
	$\Gamma \vdash e \Longrightarrow T$	*Type inference*
	$\Gamma \vdash t \in T$	*Type witnessing*

Figure 1.4: Core language.

Input: Environment Γ, test suite $\overline{t_i}$
Output: Types $\overline{T_k}$ such that $\forall i,k.\Gamma \vdash t_i \in T_k$ and $\overline{T_k}$ are likely specification types

1: TESTTOTYPE($\Gamma, \overline{t_i}$)
2: $\quad\Gamma \vdash t_i \Longrightarrow T_i$
3: $\quad T_\sqcup :=$ ANTIUNIFYALL($\overline{T_i}$)
4: $\quad G := \{T \mid T_\sqcup \sqsubseteq T \wedge$ INHABITED(T)$\}$
5: $\quad$ **return** TOPK(G)

Figure 1.5: Type inference algorithm.

We extend type inference to tests; in particular, for a test with arguments we infer a function type: $\Gamma \vdash v \to t \Longrightarrow T_1 \to T_2$ where $\Gamma \vdash v \Longrightarrow T_1$ and $\Gamma \vdash t \Longrightarrow T_2$. In this section we assume that all inferred test types are concrete (we relax this restriction in section 1.4.6). We say that a test t *witnesses* a type T in Γ ($\Gamma \vdash t \in T$), if $\Gamma \vdash t \Longrightarrow T'$ and $T' \sqsubseteq T$. The intuition is that t demonstrates a possible behavior of a function of type T, if the test's type is more specific than T. For example, the test $\Gamma \vdash$ `[1,1,1]` $\to$ `[1]` $\Longrightarrow [\texttt{Int}] \to [\texttt{Int}]$, witnesses the type $[\alpha] \to [\alpha]$.

1.4.2 From Tests to Types

Figure 1.5 presents an overview of our TESTTOTYPE inference algorithm. The algorithm takes as input an environment Γ (the component library) and a test suite $\overline{t}$, and returns a sequence of likely type specifications $\overline{T}$. Which properties need to hold of $\overline{T}$? Assume that the user's intended program is e^* and its most general type is T^* ($\Gamma \vdash e^* \Longrightarrow T^*$). Then T^* is the best type specification for synthesizing e^*: although any $T \sqsubseteq T^*$ might yield the desired program since $\Gamma \vdash e^* :: T$ necessarily holds, there might be *many more* programs e such that $\Gamma \vdash e :: T$ compared to $\Gamma \vdash e :: T^*$, hence using the more specific type as the specification is likely to yield many irrelevant results and slow down the synthesis. Of course, we do not have T^* (let alone e^*) at our disposal, so informally, the goal of TESTTOTYPE is to produce a sequence $\overline{T}$ such that T^* is likely is occur early in this sequence.

Towards this goal TESTTOTYPE proceeds in three steps. First, it uses the inference oracle to obtain the concrete types $\overline{T_i}$ of the tests. Next, it uses the function ANTIUNIFYALL to compute $T_\sqcup$, the least common generalization of $\overline{T_i}$. Then, it computes G, the set of all generalizations of $T_\sqcup$ that maybe be inhabited by relevant programs, as determined by the function INHABITED. Note that each $T \in G$ is witnessed by every test t_i in the input test suite: this is because $T_i \sqsubseteq T_\sqcup$ by the definition of least common generalization, and $T_\sqcup \sqsubseteq T$. Finally, the algorithm ranks the remaining types based on a heuristic TOPK. The remaining part of this section will introduce each step in detail.

1.4.3 Anti-Unification

Figure 1.6 details the function ANTIUNIFYALL that computes the least common generalization of a sequence of types, using *anti-unification* [93, 99]. This top-level function relies on a pairwise anti-unification procedure ANTIUNIFY, which does the actual work. At a high level, ANTIUNIFY compares the structure of two types, abstracting different substructures into fresh type variables. This function takes as input two types and returns their join, and additionally threads through an *anti-substitution* $\theta = [\overline{(T,T) \mapsto \alpha}]$—a map from pairs of types to type variables—which keeps track of the substructures that have already been abstracted.

Now, let us look at the ANTIUNIFY algorithm closely. Lines 18- 23 handle the interesting case when the top-level structure of T_1 and T_2 is different. In particular, lines 20- 23 abstract the two dissimilar types into a freshly created type variable α and add a new entry into the anti-substitution, which maps the pair (T_1, T_2) to α. To find the least common generalization, ANTIUNIFY does not always create a fresh variable when two types are different. If this pair of types is found in θ (lines 18- 19), then it has already been abstracted into some type variable α, so we simply reuse this variable. Other cases of ANTIUNIFY recursively descend into type substructures, threading the anti-substitution through. For example, when anti-unifying $[\texttt{Int}] \to [\texttt{Int}]$ with $[\texttt{Char}] \to [\texttt{Char}]$, first the two argument types are anti-unified into the type $[\alpha]$ with

Input: Sequence of concrete types $\overline{T_i}$
Output: $T_\sqcup = \sqcup(\overline{T_i})$

1: ANTIUNIFYALL(T)
2: **return** T
3: ANTIUNIFYALL($T_1; \overline{T_i}$)
4: $T :=$ ANTIUNIFYALL($\overline{T_i}$)
5: $T_\sqcup, \_ :=$ ANTIUNIFY($T_1, T, []$)
6: **return** $T_\sqcup$

Input: Types T_1, T_2, anti-substitution θ
Output: Type $T = T_1 \sqcup T_2$, anti-substitution θ

7: ANTIUNIFY($T_1 \to T_1', T_2 \to T_2', \theta$)
8: $T, \theta :=$ ANTIUNIFY(T_1, T_2, θ)
9: $T', \theta :=$ ANTIUNIFY(T_1', T_2', θ)
10: **return** $T \to T', \theta$

11: ANTIUNIFY($C\ \overline{T_i}, C\ \overline{T_i'}, \theta$)
12: **for** T_i, T_i' **do**
13: $T_i, \theta :=$ ANTIUNIFY(T_i, T_i', θ)
14: **return** $C\ \overline{T_i}, \theta$

15: ANTIUNIFY(T_1, T_2, θ)
16: **if** $T_1 = T_2$ **then**
17: **return** T_1, θ
18: **else if** $\theta[(T_1, T_2)] = \alpha$ **then**
19: **return** α, θ
20: **else**
21: $\alpha :=$ fresh type variable
22: $\theta := \theta \cup [(T_1, T_2) \mapsto \alpha]$
23: **return** α, θ

Figure 1.6: Anti-unification algorithm.

a fresh variable, and the mapping $[(\texttt{Int}, \texttt{Char}) \mapsto \alpha]$ is recorded in θ; this mapping is reused when anti-unifying the return types, in order to obtain the least common generalization $[\alpha] \to [\alpha]$ (rather than the more general $[\alpha] \to [\beta]$).

1.4.4 Type Filtering

Although the least common generalization $T_\sqcup$ computed by anti-unification reconciles the types of all tests, we also want to include more general types into the final type inference result. The challenge is that the set of all generalizations of $T_\sqcup$, $\{T \mid T_\sqcup \sqsubseteq T\}$, can contain thousands of types (see section 1.6), and we need to pick a few that are most likely to represent the user intent. Luckily, many of these types are obviously uninteresting in the sense that they can only be inhabited by meaningless programs (*i.e.* terms that ignore their arguments, or crash / diverge on all arguments). For example, the type $\texttt{Int} \to \alpha$ is uninteresting because there is no way to construct a value of arbitrary type α, while the type $\alpha \to \beta \to \beta$ is uninteresting because there is no way to use the first argument.

To filter out uninteresting types, we define a simple analysis that computes an over-

approximation of the set of inhabited types: *i.e.* if the analysis says "no", then the type can only be inhibited by degenerate terms; if the analysis says "yes", the type might still be uninhabited depending on the component library. The function INHABITED in fig. 1.7 implements this analysis. This function deems a type inhabited if its return type is *reachable* and each of its argument types is *relevant*.

Return type reachability. A return type is unreachable if it contains type variables that do not occur in the argument types (see function REACHABLE in fig. 1.7). Examples include $\texttt{Int} \to \alpha$ and $[\texttt{Int}] \to [\alpha]$. Although the latter is inhabited in the strict sense, note that all programs of this type must return the empty list regardless of the input; we consider such programs degenerate.

Argument type relevancy. An argument type is *irrelevant* if it cannot be used to compute a value of the return type (see function RELEVANT in fig. 1.7). There are two interesting cases: type variables and functions. A type variable can be used in two different ways: (1) if it directly occurs in the return type or (2) if it can be consumed by a higher-order argument, which is itself relevant. For example, the sole argument in $\alpha \to \alpha$ can be used directly, while the second argument in $(\alpha \to \beta) \to [\alpha] \to [\beta]$ can be consumed by its first argument, to eventually produce the return type. In turn, an argument of a function type $T \to T'$ is relevant if T is reachable from the rest of the arguments and T' is relevant. For example, the first argument of $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is relevant because α is reachable (from $[\alpha]$) and β is relevant (it directly occurs in $[\beta]$).

1.4.5 Type Ranking

As a final step, the function TOPK returns the k highest ranked candidate types (in our implementation $k = 10$). Our ranking approximates the likelihood that a candidate type is the user's intended type, conditioned on the examples provided. At a high level our strategy approximating that likelihood first picks the "simplest" types given the tests, then picks the most general types. Our ranking assumes the user's tests were just informative enough, that any type

Input: Type T
Output: May T be inhabited?

```
1: INHABITED(T)
2:     (A,R) := ARGSRET(T)
3:     reach := REACHABLE(A,R)
4:     rel := ⋀_{T∈A} RELEVANT(A,R,T)
5:     return reach ∧ rel
```

```
1: ARGSRET(T → T')
2:     (A,R) := ARGSRET(T')
3:     return ({T} ∪ A, R)
4: ARGSRET(T)
5:     return ({}, T)
```

```
1: FUNTYPES(T → T')
2:     return T → T'
3: FUNTYPES(C T̄)
4:     return ⋃ FUNTYPES(T)
5: FUNTYPES(α)
6:     return ∅
```

Input: Argument types A, return type R
Output: Whether R can be computed from A

```
1: REACHABLE(A,R)
2:     return ⋃ TVARS(A) ⊇ TVARS(R)
```

Input: Argument types A, return type R, type T
Output: Whether T can be used to compute R

```
1: RELEVANT(A,R,α)
2:     if α ∈ TVARS(R) then
3:         return true
4:     for T_a ∈ A, T ∈ FUNTYPES(T_a) do
5:         (A',R') := ARGSRET(T)
6:         if REACHABLE(A',α) ∧
7:             RELEVANT(A \ T_a,R,T) then
8:             return true
9:     return false
10: RELEVANT(A,R,T → T')
11:     return REACHABLE(A,T) ∧
12:     RELEVANT(A ∪ {T'},R,T')
13: RELEVANT(A,R,T)
14:     return true
```

Figure 1.7: Type filtering algorithm.

structures or similarities were intentional. The function is based on lexicographic ordering of three simple heuristics.

Our *first heuristic* penalizes generalizations that abstract over a complex type: a function or a non-nullary constructor application. For example, consider possible generalizations of $[\texttt{Int}] \to [\texttt{Int}]$. This heuristic penalizes abstracting this type into $\alpha \to \alpha$ or α, because these generalizations abstract over a list constructor and a function, respectively. The intuition is that a user is unlikely to supply a complex value if it is not required to illustrate the behavior: *e.g.* it is more natural to illustrate the identity function with the test $1 \to 1$ rather than $[1,1] \to [1,1]$. As an optimization, our implementation does not generate this kind of generalizations in the first place, since in practice they never make it into top k. We make an exception for the type $[\texttt{Char}]$ and do not penalize abstracting it into α, because of the special string literal syntax, which makes values of this type appear simple.

Our *second heuristic* is to prioritize types that generalize same substructures into the same variable. Going back to the $[\texttt{Int}] \to [\texttt{Int}]$ example, the generalization $[\alpha] \to [\alpha]$ has higher rank than $[\alpha] \to [\texttt{Int}]$ because the former abstracts both occurrences of $\texttt{Int}$ into α. We assume a-priori that simpler types, those with fewer distinct atomic types, are more likely than complex types, with more atomic types, for reusable code snippets HOOGLE+ is capable of producing. To implement this heuristic, we build an inverse substitution between the anti-unification result $T_\sqcup$ and the generalized type T, and penalize T proportionally to the size of this substitution. In our example, the inverse substitution for $[\alpha] \to [\alpha]$ is $[\texttt{Int} \mapsto \alpha]$, whereas for $[\alpha] \to [\texttt{Int}]$ it is $[\texttt{Int} \mapsto \{\alpha, \texttt{Int}\}]$, so the former is ranked higher (note that we keep the identity mapping $\texttt{Int} \mapsto \texttt{Int}$ in the substitution, unless all occurrences of $\texttt{Int}$ were replaced).

Our *third heuristic* is to prioritize general types over specific types. In our example, $[\alpha] \to [\alpha]$ has higher rank than $[\texttt{Int}] \to [\texttt{Int}]$ because their inverse substitutions have the same size one, but the former is more general. This heuristic easily over-generalizes: in the absence of the second heuristic, it prefers $[\alpha] \to [\beta]$ on our example. For this reason we give it the least

priority.

1.4.6 Support for Ambiguously-Typed Tests

Our formalization so far assumed that each test t_i has a unique concrete type T_i. Unfortunately, this is not always the case: Haskell values can have polymorphic types, and using such values inside tests presents a subtle issue. The simplest example of a polymorphic value is the empty list; so, what is the type of the test `[]` $\rightarrow$ 0? The user could have intended $[\alpha] \rightarrow$ `Int` (*e.g.* list length), `[Int]` $\rightarrow$ `Int` (*e.g.* sum of the elements), or even `[Char]` $\rightarrow$ `Int` (*e.g.* number of spaces). Note that we cannot assume that the $T_{\sqcup}$ type for this (singleton) test suite is $[\alpha] \rightarrow$ `Int` with α interpreted as universally quantified, because this would preclude the inference of the other two plausible type specifications. Polymorphic values are not a corner case that can simply be ignored: values like `[]` and `Nothing` are common enough, but things get even worse with higher-order tests, because many functions are naturally polymorphic. For example, consider the following test for the function `applyNTimes` from our user study, which applies a function to some initial value n times (see section 1.7 for details): `(\x -> x ++ x) -> "s" -> 2 -> "ssss"`. Here the first argument has a polymorphic type $[\alpha] \rightarrow [\alpha]$, and, perhaps counter-intuitively, the test does not actually constrain α to be `Char`.

In order to support ambiguously-typed tests, we extend the syntax of types with a separate kind of type variables that we refer to as *wildcards*: $T ::= \ldots \mid ?\alpha$. The wildcards are introduced by the inference oracle for tests $\Gamma \vdash t \Longrightarrow T$, when tests contain polymorphic values. For example, we infer the type $[?\alpha] \rightarrow$ `Int` for `[]` $\rightarrow 0$. Unlike regular type variables α, which are implicitly universally quantified, a wildcard stands for a concrete type a user had in mind, which is unknown to the synthesizer.

To accommodate for wildcards during type specification inference, we need to modify to the function ANTIUNIFY. The join of two types is now not a single type but a set

Input: Types T_1, T_2, anti-substitution θ, dis-unification constraints Ω
Output: Set of types $\overline{T_i}$ such that $T_i = T_1 \sqcup T_2$, and their respective anti-substitution θ, wildcard substitution $?\sigma$, and dis-unification constraints Ω

1: ANTIUNIFY(T, T, θ, Ω)
2: **return** $\{T, \theta, [], \Omega\}$

3: ANTIUNIFY$(T_1 \to T_1', T_2 \to T_2', \theta, \Omega)$
4: $T, \theta, ?\sigma, \Omega \leftarrow$ ANTIUNIFY$(T_1, T_2, \theta, \Omega)$
5: $T', \theta, ?\sigma, \Omega \leftarrow$ ANTIUNIFY$(?\sigma T_1', ?\sigma T_2', \theta, \Omega)$
6: **return** $\{T \to T', \theta, ?\sigma, \Omega\}$

7: ANTIUNIFY$(C\ \overline{T_i}, C\ \overline{T_i'}, \theta, \Omega)$
8: **for** T_i, T_i' **do**
9: $T_i, \theta, ?\sigma, \Omega \leftarrow$ ANTIUNIFY$(?\sigma T_i, ?\sigma T_i', \theta, \Omega)$
10: **return** $\{C\ \overline{T_i}, \theta, ?\sigma, \Omega\}$

11: ANTIUNIFY$(?\alpha, T_2, \theta, \Omega)$
12: $R :=$ ABSTRACT$(?\alpha, T_2, \theta, \Omega)$
13: $?\sigma := [?\alpha \mapsto T_2]$
14: **return** $R \cup \{T_2, ?\sigma\theta, ?\sigma, ?\sigma\Omega \mid \mathsf{sat}(?\sigma\Omega)\}$

15: ANTIUNIFY$(T_1, ?\alpha, \theta, \Omega)$
16: ... (symmetrical)

17: ANTIUNIFY$(T_1, T_2, \theta, \Omega)$
18: **return** ABSTRACT$(T_1, T_2, \theta, \Omega)$

Input: Types T_1, T_2, anti-substitution θ, dis-unification constraints Ω
Output: Type variables $\overline{\alpha}$ and corresponding anti-sub θ, sub $?\sigma$, new constraints Ω

19: ABSTRACT$(T_1, T_2, \theta, \Omega)$
20: $\alpha :=$ fresh type variable
21: $\theta' := \theta \cup [(T_1, T_2) \mapsto \alpha]$
22: $\Omega' := \Omega \cup \{(T_1, T_2) \not\sim (T_1', T_2')] \mid [(T_1', T_2') \mapsto \alpha] \in \theta\}$
23: $R := \{\alpha, \theta', [], \Omega' \mid \mathsf{sat}(\Omega')\}$

24: **for** $[(T_1', T_2') \mapsto \alpha] \in \theta$
25: s.t. $\exists ?\sigma = \mathsf{mgu}((T_1', T_2'), (T_1, T_2))$
26: and $\mathsf{sat}(?\sigma\Omega)$ **do**
27: $R := R \cup \{\alpha, ?\sigma\theta, ?\sigma, ?\sigma\Omega\}$
28: **return** R

Figure 1.8: Anti-unification algorithm with wildcard type variables.

of types, for each possible instantiation of the wildcard. For example, consider two tests for the function `applyNTimes: (\x → x ++ x) → "s" → 2 → "ssss"` and `(\x → 0:x) → [1] → 3 → [0,0,0,1]` whose types are, respectively, $([?\alpha] \to [?\alpha]) \to [\texttt{Char}] \to \texttt{Int} \to [\texttt{Char}]$ and $(\texttt{[Int]} \to \texttt{[Int]}) \to \texttt{[Int]} \to \texttt{Int} \to \texttt{[Int]}$. The join of these two types is a pair of types $([\beta] \to [\beta]) \to [\beta] \to \texttt{Int} \to [\beta]$ and $(\texttt{[Int]} \to \texttt{[Int]}) \to [\beta] \to \texttt{Int} \to [\beta]$. The first result comes from instantiating $?\alpha \mapsto \texttt{Char}$ and the second one from $?\alpha \mapsto \texttt{Int}$; importantly, any other instantiation would lead to a type that is more general than either of the two. After computing the join, the rest of the inference algorithm proceeds by taking the union of all generalizations of each member of the join, and performs the filtering and ranking as before.

Anti-Unification with Wildcards. Our algorithm for computing the join efficiently (without enumerating infinitely many potential wildcard instantiations) is shown in fig. 1.8. There are several differences between this algorithm and ANTIUNIFY from fig. 1.6. First, the algorithm returns a *set* of anti-unification results, and uses Haskell-like monadic notation to compute all combinations of results in lines 3–6 and 7–10. Second, each anti-unification result also includes a wildcard substitution $?\sigma$, which maps wildcards to types, and a set of *dis-unification constraints* Ω of the form $T_1 \not\sim T_2$. These components record the decisions made about wildcard instantiations in the current branch of the search.

The most interesting case of anti-unification is captured by the helper procedure ABSTRACT, which abstracts two types with dissimilar top-level structure into an anti-unification variable. Because the input types T_1 and T_2 might contain wildcards, ABSTRACT cannot decide a-priori whether to create a fresh anti-unification variable or to reuse one of the variables already in θ. Instead it tries *all of these option* in turn and discard those that conflict with the accumulated dis-unification constraints Ω. Lines 20-23 create a fresh type variable, and add to Ω the constraints that the input types T_1, T_2 must not unify with any existing key in θ; if the resulting constraints Ω are unsatisfiable (*i.e.* contain a dis-unification of equal types), then this option

is discarded. Lines 24–27 instead attempt to reuse an existing mapping $[(T_1', T_2') \mapsto \alpha]$ from θ; the mapping is only considered if (T_1, T_2) unifies with the key (T_1', T_2') (mgu stands for "most general unifier"), and the result of this unification is consistent with the current Ω. Note that in the absence of wildcards, ABSTRACT reduces exactly to the case of dissimilar types in fig. 1.6: in this case, the pair (T_1, T_2) either occurs as a key in θ exactly or it does not; hence only one set of dis-unification constraints computed in lines 22 and 26 can be satisfiable, and ABSTRACT will always return exactly one result.

1.4.7 Support for Type Classes

Type classes are a popular feature of the Haskell type system [122], and we support them by making another modification to ANTIUNIFY. When abstracting types T_1 and T_2 into a fresh type variable, we compute the set of type classes these two types have in common and attach a corresponding type class constraint to the resulting variable. For instance, consider the anti-unification of $[\texttt{Int}] \to [\texttt{Int}]$ and $[\texttt{Bool}] \to [\texttt{Bool}]$. When the first pair $(\texttt{Int}, \texttt{Bool})$ is anti-unified, we check that both `Int` and `Bool` are instances of the type classes `Eq` and `Ord`. Hence, we abstract them into constrained type variable $(\texttt{Eq}\ \alpha, \texttt{Ord}\ \alpha) \Rightarrow \alpha$; collecting constraints on all variables, we compute the anti-unifier $(\texttt{Eq}\ \alpha, \texttt{Ord}\ \alpha) \Rightarrow [\alpha] \to [\alpha]$ for the top-level types.

1.5 Tests for Elimination and Comprehension

Next, we describe how we use the SMALLCHECK's property-based testing [101] to eliminate undesirable candidates and produce examples that aid different comprehension goals.

1.5.1 Elimination

The *elimination* procedure takes as input a sequence P of candidate programs found by the synthesizer and returns a subsequence $P^* \subseteq P$ that only contains *meaningful* and *unique*

programs.

Meaningful Programs. A candidate program is *meaningful* if there *exist* an input value on which the candidate terminates and produces an output value within some timeout. Formally, we denote the output of a program p on an input tuple i as $[\![p]\!](i)$, where $[\![p]\!](i) = \bot$ if p crashes or diverges on i. We say that a program is meaningless if $\forall i.\ [\![p]\!](i) = \bot$. For example, the well-typed candidate `\x → head []` is meaningless as it yields $\bot$ regardless of the input `x`.

Testing Meaningfulness. We cannot test exactly whether an arbitrary program p is meaningless for two reasons: first, the set of possible inputs can be infinite, and second, for a given input we might need to wait an unbounded amount of time to determine whether a program terminates. Instead we say that p is *likely meaningless* with respect to a finite set of inputs I and timeout T if $\forall i \in I.\ [\![p]\!]^T(i) = \bot$, where $[\![p]\!]^T$ denotes the result of executing p for at most time T. HOOGLE+ tests whether a candidate is likely meaningless by invoking SMALLCHECK to enumerate all the values of a given input type up to a given constructor depth, and then running the candidate on the enumerated inputs to check if they successfully produce an output within a given timeout. Thus, candidates like `\x → head []` that yield $\bot$ for all inputs are deemed meaningless and eliminated. Because the check is approximate, HOOGLE+ might erroneously eliminate a meaningful program if it requires large inputs to produce an output. Our empirical evaluation shows (section 1.6), that this happens very rarely in practice.

Lazy Candidates Can be Meaningful. In a lazy language like Haskell, determining whether a given program output $[\![p]\!]^T(i)$ is $\bot$ is actually non-trivial. Generally, HOOGLE+ has to *force* program evaluation, for example, by *printing* the output (*i.e.* converting it to string). While doing so, however, HOOGLE+ has to take special care not to eliminate programs that return infinite data structures. For example, consider the candidate `\x → repeat x` which returns an infinite list of `x` values. This candidate should be deemed meaningful, since it is common practice to produce infinite data structures that can be consumed lazily. Printing the output of this program, however, leads to a non-terminating execution, and hence by default the program

is deemed meaningless.

To overcome this challenge, we use a special function `approxShow` introduced in [25], which prints an execution result only up to a finite given depth. If the result can be partially printed, the program is deemed meaningful. In the example above we invoke `approxShow 3 (repeat 1)` to print the result of `repeat 1` up to depth 3, which yields `"[1,1,1,_"`. As this value is not $\perp$, HOOGLE+ deems the candidate to be meaningful.

Unique Programs. We say that a candidate p is *observationally equivalent* to another candidate p', written $p \equiv p'$ if $\forall i.\ [\![p]\!](i) = [\![p']\!](i)$, *i.e.* if p and p' return the same results for all inputs i. We say a candidate p is *unique* with respect to a set of candidates P' if for each $p' \in P'$ we have $p \not\equiv p'$, *i.e.* if for each p' there exists some *distinguishing input* i such that $[\![p]\!](i) \neq [\![p']\!](i)$.

Testing Uniqueness. Just like meaningfulness, uniqueness is impossible to check exactly. Instead, we say that p is a *likely duplicate* with respect to P' and relative to an input set I and timeout T, if $\exists p' \in P'.\forall i \in I.[\![p]\!]^T(i) = [\![p']\!]^T(i)$, *i.e.* there exists a program p' such that on any input from I either both programs return the same value or they both fail (crash or execute longer than T).

HOOGLE+ presents the candidates to the user one-by-one, as soon as they are found. For each new candidate p, HOOGLE+ uses SMALLCHECK to test whether it is a likely duplicate with respect to the programs $p_1, \ldots, p_k$ that were previously shown to the user. Because the check is approximate, HOOGLE+ might accidentally eliminate a unique program if the distinguishing input required to differentiate it from every previous program is large; again, our study show that this rarely happens in practice.

Examples of Unique Programs. Our uniqueness test yielded some interesting results. Consider a queries `applyNTimes :: (a → a) → Int → a → a` from our user study, which composes n copies of a given function and applies it to an initial value. To our surprise, HOOGLE+ synthesized two candidate solutions,

which at the first glance appeared equivalent: `\f n x → (iterate f x) !! n` and `\f n x → foldr ($) x (replicate n f)`. Closer examination revealed, that in fact, the two terms above behave identically when `n` is non-negative, but when `n` is negative, the former solution crashes while the latter returns `x`. On the other hand, consider the result found by the query `applyPair :: (a → b, a) → b` which applies the first element in the pair to the second element. HOOGLE+ returned the expected solution `\p → (fst p) $ (snd p)` but we found that the uniqueness test eliminated a seemingly different solution, `\p → uncurry id p`. Upon closer examination, however, we found that these two candidates indeed have the same behavior.

1.5.2 Comprehension

Often, the best way to understand a piece of code is to run it on some inputs, observe the outputs and then build a mental model relating the two. However, to understand a new piece of code, one does not typically run arbitrarily (randomly) chosen inputs. Instead, we can often discern patterns from small, carefully chosen inputs, that may be crafted to demonstrate some difference between the program under study and another candidate.

Examples for Comprehension. An *example* for a program p is a pair of input and output values (i, o) where $o = [\![p]\!](i)$. Motivated by the above observations, HOOGLE+ generates three kinds of examples to comprehend the synthesized programs more easily, deeply, and rapidly.

1. ***Meaningfulness*** : HOOGLE+ determines that the program is meaningful by finding at least one *success* example (i_{succ}, o_{succ}) where $o_{succ} \neq \bot$ If p is a partial function, then in the course of determining meaningfulness HOOGLE+ may also have found a failure example $(i_{fail}, \bot)$. Both the success and failure examples are shown to the user to help with comprehension.

2. ***Uniqueness*** : Additionally, HOOGLE+ only shows programs that are unique with respect

to all previously shown candidates. This is established by a set of *uniqueness* examples $\overline{(i_j, o_j)}$ that differentiate p from its predecessors P', in that for each $p'_j \in P'$, we have $o_j \neq [\![p_j]\!](i_j)$. Thus, each of these uniqueness examples are also shown to help the user understand how the candidate p is different than the other p'_j candidates.

3. ***Functionality*** : Finally, sometimes the user wants other examples that illustrate the functionality of the candidate. Hence, HOOGLE+ generates a set of functionality examples where each new input is different from *all* previously generated inputs.

1.6 Empirical Evaluation

In this section, we empirically evaluate the effectiveness of type inference from tests and elimination.

Benchmarks. In all experiments, we use the component library and benchmark suite used by [46]. Each of these benchmarks is a type-only query. We exclude one benchmark, which contains the type `ByteString`, since it is impossible to provide a test value for this type in HOOGLE+ (this type requires a special function call to convert from a string). To the remaining set of queries we add the tasks from our user study (section 1.7), arriving at a total of 45 benchmarks.

1.6.1 Type Inference From Tests

We evaluate the quality of the type inference algorithm on two sets of inputs: tests written by participants in our user study, and tests generated randomly by QUICKCHECK.

User-Provided Tests. Our first experiment evaluates the accuracy rate of type inference algorithm on real user data. For this purpose, we consider the five user study tasks, for which the correct type is defined in the study definition (see subsection 1.7.1). We collected 76 type

inference queries for these tasks out of the logs of searches performed by users in the course of the user study, after ruling out ill-formed searches (e.g., syntactically incorrect examples). We ran the type inference algorithm on these queries. In 39 queries the correct answer is ranked first, in 4 queries it is ranked second, and in one query it is ranked third. The median rank of all queries is 1. For only 5 out of 76 queries the correct result does not appear in the top 10. This shows that our algorithm infers correct types from user-provided tests.

Randomly Generated Tests. While HOOGLE+ effectively infers types for tasks from our user study, this only accounts for 5 out of 45 benchmarks. Thus, we perform a second experiment to determine whether our inference algorithm generalizes to *other* programming tasks. Recall that each of our benchmarks is a type-only query. In our second experiment, we use QUICKCHECK [23] to generate random input-output examples as follows. First, if the query has type parameters, we randomly instantiate them using a fixed set of base types (*e.g.* `Int`, `Char`, *etc*), to get a randomly generated monomorphic instantiation. Next, we invoke QUICKCHECK on the instance to generate values for the inputs and outputs of the signature to get a concrete test for the original type query. We evaluate our inference algorithm by running it on one, two, or three randomly generate tests and measuring the rank at which the "correct" signature (*i.e.* original type query) appears in the inference results. We report average results over six runs to reduce the uncertainty of random example generation. The results of this experiment are summarized in Figure 1.9 (Left).

Results. The heat map is sectioned by the number of type variables in benchmark queries, and each cell of the heat map shows the percentage of benchmarks (of that number of type variables) where the correct result appears at that rank. Cells with darker colors represent a larger percentage.

For the most part, HOOGLE+ ranks the correct solution first or second, across the board. The few exceptions are seen at the bottom right of the chart, in runs with four type variables and only one test, making it hard to get the correct generalization from single concrete type.

Within a given number of type variables, the rank of the correct type worsens as the number of tests decreases. This is as expected as fewer tests and more type variables lead to a larger set of possible generalizations, which makes it harder to identify the correct ones.

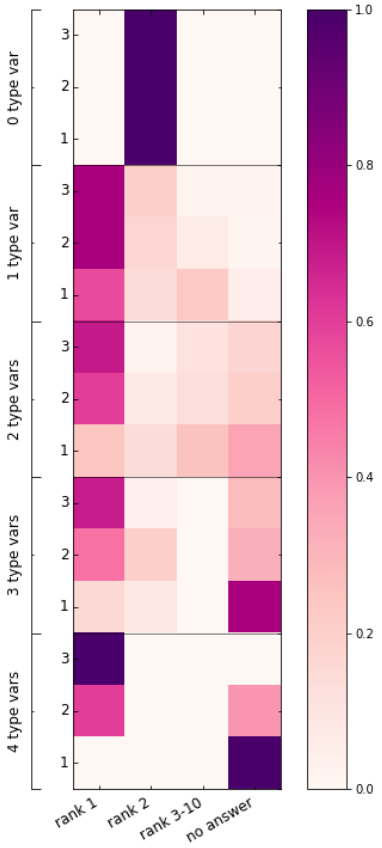
To confirm this intuition, we study the effect of the number of type variables and tests on the number of generalization, before and after filtering. Figure 1.9 (Right) shows the minimum and maximum numbers of generalizations as well as median ranks over six runs. As we can see, the maximum value of type generalizations may reach hundreds of thousands or even millions in the case of many type variables when few test inputs are provided. However, our inference algorithm still produces the correct solution at a high rank: it has a median rank 1 or 2 in 12 out of 15 cases.

The difference in pre- and post-filtering generalizations shows that our filtering algorithm is highly effective, usually reducing the number of generalizations by at least an order of magnitude. This drastically reduced search space is a big step toward selecting the correct program. Of course, there is room for improvement as in a few cases (*e.g.* 3 type variables and 1 test) we still fail to provide a good answer.

1.6.2 Elimination

Next, we evaluate our test-based technique for eliminating irrelevant program candidates returned by TYGAR. Recall that test-based elimination can produce *false negatives*, *i.e.* erroneously eliminate a meaningful and unique program because it did not search large enough inputs or did not wait long enough for the program to terminate. On the other hand, elimination cannot produce *false positives*: if it deems a program meaningful and unique, this is always accurate. In the rest of the section we evaluate both the *importance* of elimination (the number of true negatives) and its *recall* (the proportion of false negatives).

Experimental setup. To evaluate the elimination strategy in HOOGLE+, we ran three experiments on the 45 benchmarks in our suite:



# vars	# tests	median rank	pre-filter generalizations min	pre-filter generalizations max	post-filter generalizations min	post-filter generalizations max
0	3	2	45	2,216	7	413
0	2	2	45	2,216	7	413
0	1	2	45	2,216	7	413
1	3	1	9	124,176	3	37,703
1	2	1	6	3,329,455	3	779,819
1	1	1	27	3,329,455	9	779,819
2	3	1	9	40,728	6	5,071
2	2	1	9	141,127	6	31,832
2	1	6	27	407,759	18	37,778
3	3	1	2,430	546,966	496	71,853
3	2	2	2,430	969,220	54	97,433
3	1	-	35,964	8,242,171	4,846	589,284
4	3	1	243,000	818,262	38,520	77,514
4	2	1	414,558	2,276,910	27,759	202,652
4	1	-	3,314,300	81,100,863	168,838	5,939,433

Figure 1.9: Type inference results on random generated tests. We compare the results between different number of type variables in the expected type and different number of tests. The x-axis is different ranks. The y-axis is grouped by number of type variables, and each group has three rows corresponding to evaluation on 3 to 1 tests. Each cell shows the row-wise percent of benchmarks. Darker colors mean more benchmarks have correct answers of that rank among those with the same number of vars and tests. (Right) Type inference counts on random generated tests. We report the median rank of correct solutions, min and max counts of type generalizations before and after filtering. '-' in ranks means no correct answer found in the top 10 results.

1. TYGAR-180: we ran TYGAR with a 180-second timeout per benchmark

2. HP-180: we ran HOOGLE+ with a 180-second timeout per benchmark

3. HP-360: we ran HOOGLE+ with a 360-second timeout per benchmark

We then manually labeled all meaningless results in TYGAR-180 and partitioned the rest into semantic equivalence classes. Next, we compared results in HP-180 to our labeled set, expecting one representative from each meaningful equivalence class to remain. We observed some differences between HP-180 and the labeled set; we refer to these mistakenly discarded programs as *loss due to misclassification*. Finally, we compared results in HP-180 with those in HP-360, detecting programs that are missing simply because they take too long to generate with elimination; we refer to this as *loss due to testing overhead*. The results are shown in Figure 1.10.

The graph shows that the number of true negatives is often high, sometimes an order of magnitude higher than the number of true positives. Hence we conclude that elimination is important: without it, the user is likely to be overwhelmed with meaningless and redundant programs when searching with a type-only specification.

Loss due to Misclassification. Programs lost by misclassification are programs where no witness to their meaningfulness or uniqueness was found. When looking for a witness, we only enumerate examples up to a certain constructor depth (in this experiment we used depth 3) within a timeout of 4 seconds. When the witness is outside this range, HOOGLE+ will misclassify the program as uninteresting.

Our results show that misclassification is infrequent. In benchmarks with relatively high misclassification rates (*e.g.* `flatten`) it is caused by the complexity of input types, which in turn requires large constructor depths to generate interesting inputs. For instance, two solutions for the query `flatten :: [[[a]]] → [a]` are `\xs → concat (init xs)` and `\xs → concat (concat xs)`, with a distinguishing input `xs = [[[]], [[0]]]`; this input, however, lies at constructor depth 5, and hence is not generated.

Loss due to Testing Overhead. It takes extra time for HOOGLE+ to test meaningfulness and uniqueness for each candidate, which in turn takes away from the time to perform the TYGAR candidate search. This means TYGAR may find fewer results than before. Most benchmarks have a testing overhead loss rate of no more than 10%.

We also carefully examined the benchmarks with high testing overhead loss rates (*e.g.* `takeNdropM`), and found that they all have a large number of displayed candidates found by the synthesizer. This means it takes more time to establish uniqueness for each *new* candidate as we must find inputs that distinguish the candidate from *each* previously displayed one. This delay in uniqueness check prevents the synthesizer from enumerating more candidates yielding testing overhead losses.

Trade-off. There is a trade-off between misclassification rate and testing overhead: increasing constructor depth and testing timeout makes test-based elimination more precise and thus decreases loss due to misclassification; at the same time, this increases testing overhead and the associated loss. We experimented with different timeouts and depth limits, and found that changing these parameters had no significant effect on most benchmarks.

1.7 User Study

We conducted a user study that sought to answer questions about the utility and usability of our tool. We focused on the following research questions:

- **RQ1**: *Does synthesis help programmers solve program search tasks compared to traditional methods?* We believe that better performance on search tasks leads to greater productivity, as many mundane programming tasks boil down to snippet search.
- **RQ2**: *How do functional programmers express their intent in the synthesizer?* What styles of input do these programmers use to guide their search with a tool? Do they prefer to search with types, tests, or a mix?

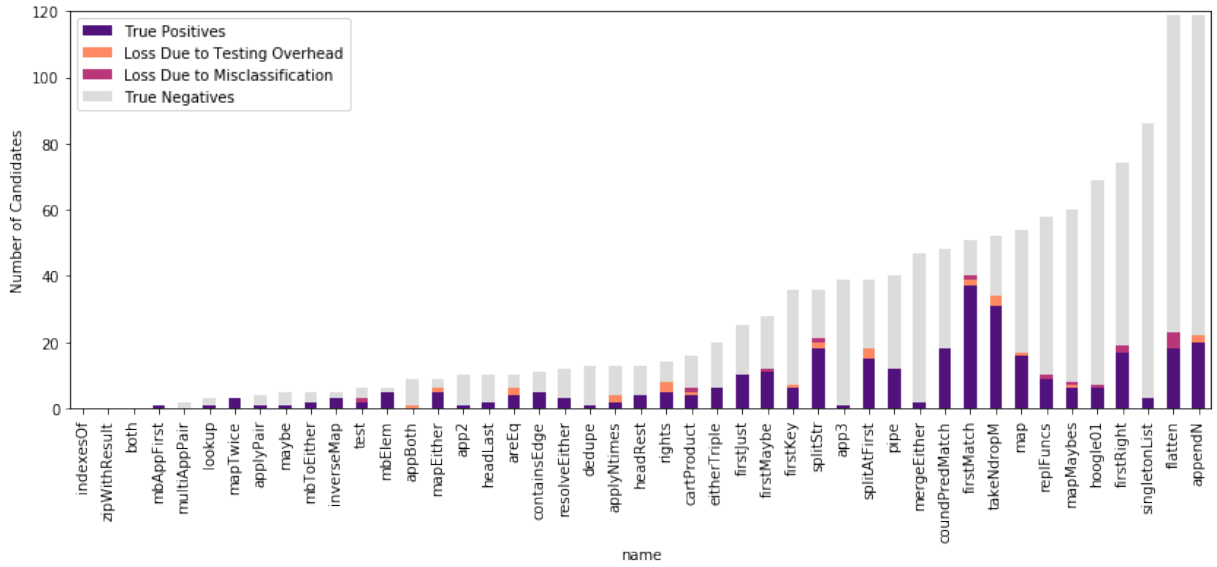


Figure 1.10: Elimination results on the benchmark suite. For each benchmark we report the number of true positives (interesting programs that are reported), true negatives (uninteresting programs that are eliminated), and false negatives (programs that are mistakenly eliminated or never generated due to testing overhead).

- **RQ3**: *How do functional programmers interpret the results they receive from the synthesizer?* Users have several methods to understand a candidate presented to them. HOOGLE+ provides documentation, automatically generated examples, user provided examples, and the code itself. Out of this wealth of information, what did programmers find useful in understanding the programs they are looking at and making decisions about candidate programs?

Choosing a control. In order to better understand the way Haskell programmers search for code today, we performed an initial information-gathering survey on the way Haskell programmers search for code. We surveyed 151 people online. Of those respondents who use Haskell in varied settings (47% industry, 48% academic, 54% open source are the top three) and with different levels of experience (12% less than one year, 29% 1-6 years, and the remainder over 7 years), 84 users listed HOOGLE as their first engine of choice and further 27 as their second engine of choice for Haskell code. HOOGLE permits searching for a library function by either a type signature or by its name. Of those who listed HOOGLE as one of their top choices, 121 listed searching by type and 107 listed searching by name as one of their preferred search

modalities. The next most popular search engine, Google, was reported by only 37 users as their their top choice. We therefore assess the utility of our method compared to the most frequently used alternative, and choose searching with HOOGLE as our control.

1.7.1 Study Design

Recruitment. The Haskell community is scattered in small pockets around the world. We planned our study to work remotely to sample from the broad community. We recruited 30 participants (6 female, 24 male) via Twitter, Reddit, university lab mailing lists, and mailing lists devoted to functional programming or specifically Haskell. 22 participants were from academia (11 different institutions) and 8 were from industry (7 different companies). We asked participants to self-identify with same experience classification from our exploratory survey, and did not admit into the experiment users who have never used Haskell regularly. Of those categories, we had 12 participants new to Haskell, 10 intermediate-level users, and 8 expert users. The participants were paid for their time.

Task Selection. We selected our tasks to test different aspects of Haskell that programmers must keep in mind when searching for program snippets. We created two tasks that require using a higher-order function, while the other two tasks do not need a function as an argument. Their full description as provided to users can be found in Figure 1.11:

0. Training - `concatNTimes :: Int → [a] → [a]`. This program concatenates its second argument `n` times to itself. This task was intended to be simple with no express challenges.
   Solution: `\i xs → concat (replicate i xs)`

1. Task A - `firstJust :: a → [Maybe a] → a`, gets the first `Just` from the list with a fallback, default value. This task is challenging as it requires composing three

uncommon components.

Solution: `\def xs → fromMaybe def (listToMaybe (catMaybes xs))`

2. Task B - `dedup :: Eq a => [a] → [a]`, our running example removes adjacent duplicates from its input. This task challenges participants to consider and produce a typeclass constraint. Solution: `\xs → map head (group xs)`

3. Task C - `applyNTimes :: (a → a) → Int → a → a`, applies its function argument `n` times to its last argument. This task requires thinking about combining higher-order functions. Solution: `\f i x → (iterate f x) !! i` or `\f n x → foldr ($) x (replicate n f)`

4. Task D - `inverseMap :: [a → b] → a → [b]`, applies each element of its list of functions to its second argument. Like task C, this also requires considering higher-order functions. Solution: `\fs x → zipWith ($) fs (repeat x)` or `\fs x → map ($ x) fs`

Procedure. Each participant was asked to complete four short program search tasks, listed above. Each of the four tasks had a high level, English-language description of the desired result, along with one example to characterize the expected results of that program, as shown in Figure 1.11. The first two tasks were completed under our control workflow, and the next two tasks—under the treatment workflow with HOOGLE+. Each half of the study opened with a training task to allow the participant some time to familiarize themselves with the workflow; each half closed with a short questionnaire. Each task was time limited to 8 minutes to ensure the whole study would fit within one hour.

Control. In the control segment of the experiment, users were provided with an online GHCi session[3] and the HOOGLE search engine, which they were permitted to search by name

[3] `https://repl.it/languages/haskell`

Training	***Description.*** Function `concatNTimes` takes two inputs, a natural number n and a list xs. It concatenates xs n times to itself. ***Example.*** `concatNTimes 2 "abc" = "abcabc"`
Task A	***Description.*** Function `firstJust` takes two arguments: a list of Maybe a's and a default value. It returns the first element from the list that is a Just or the default, if no such element exists. ***Example.*** `firstJust 0 [Nothing, Just 1] = 1`
Task B	***Description.*** Function `dedupe` takes one input, a list. It returns the list with any adjacent duplicate values removed. ***Example.*** `dedupe "aaabbbab" = "abab"`
Task C	***Description.*** Function `applyNTimes` takes three arguments: a one-argument function f, a natural number n, and an initial value x. It applies f to x, n times, setting up a pipeline of function applications. ***Example.*** `applyNTimes (\x → x ++ x) 3 "f-" = "f-f-f-f-f-f-f-f-"`
Task D	***Description.*** Function `inverseMap` takes two inputs, a list of functions fs and an input x. It applies each element of fs to x and returns a list of those results. ***Example.*** `inverseMap [(\x → x + 2), (\x → x * 2)] 5 = [7, 10]`

Figure 1.11: The task names and descriptions provided to users in our study.

or by type. The GHCi session was pre-seeded with all the same function and modules that HOOGLE+ had at its disposal. Users were instructed to solve the task with a composition of existing library functions.

The purpose of the interpreter was to compose the different components of the solution. We therefore imposed several restrictions to focus users on program search: (1) Participants could not invoke GHCi's type informational features on library functions such as `:t`–which prints the type of an expression– `:i` or `:browse`–which give further information on a type or module; (2) they could not import any additional modules, and (3) they could only invoke GHCi to execute a (partial) solution on an example input or to inquire about the type of their (partial) solution. Additionally, users were not allowed to use control structures, recursion, or pattern matching in their solution to ensure a component-based answer.

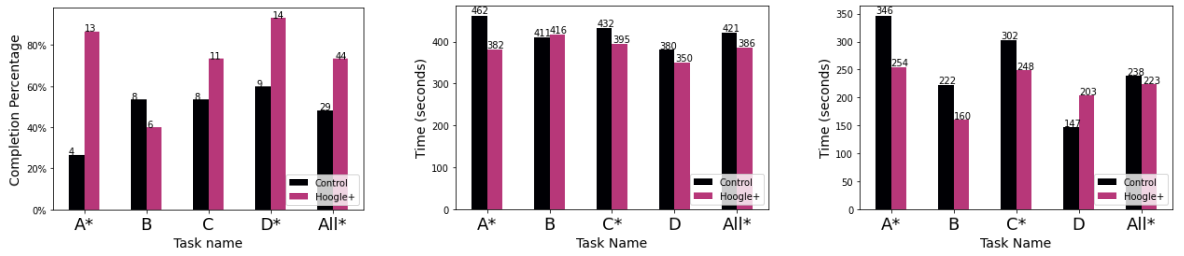
Users could follow any links on the HOOGLE website, but were forbidden from making an open-internet search (*e.g.* Google or Stackoverflow). Participants were given a training task to familiarize themselves with these interaction restrictions.

Treatment. Users were presented with our tool, as presented in section 1.2–they did not have access to the GHCi or to HOOGLE. Users were trained with the same training task as in the first half of the study.

Experiment groups. Every participant in our study executed the control setting, followed by the treatment (within-subjects). In order to collect data on all tasks, we assigned users to one of two groups, rotating which tasks are control tasks and which are treatment tasks. Note that we did not additionally randomize the order of the tasks, since our control setting is similar to users' regular workflow, so there is no need to isolate knowledge transfer from it.

The tasks were grouped together: task group 1, task A and task C; and task group 2, task B and task D. We grouped the tasks as A/C and B/D to ensure that each group would have one higher-order query and one first-order query. The study groups are then:

1. Task group 1 in control, then task group 2 with HOOGLE+;



(a) Task completion percentage. Data labels are absolute completions.

(b) Task time averages (seconds), including timeouts

(c) Task time averages (seconds), without timed out sessions

Figure 1.12: Comparison of time to complete, with and without participant timeouts. 1.12a shows completion improvements. An asterisks next to a task indicates a statistically significant change.

2. Task group 2 in control, then task group 1 with HOOGLE+.

Users were randomly assigned into one of the two study groups, while preserving an equal distribution of experience between the groups. Each group had: 6 with less than 1 year, 5 with 1-6 years experience, and 4 with 7+ years experience.

1.7.2 Results

We present the results relevant to each research question separately. In the remainder of this section, we set the threshold for statistical significance at $p < 0.1$.

RQ1: Does synthesis help programmers with program search tasks?

For each of the four tasks in a session we measured the time until the user completed the task, and whether the task was completed or timed out (8 minutes). The results are shown in Figure 1.12.

Completion rates. Of the 60 tasks attempted with each tool, 29 were completed with HOOGLE and 44 with HOOGLE+, a 51% increase in completion rate with HOOGLE+. Figure 1.12a shows the breakdown by task.

In a per-task breakdown, completion rates of users improved for tasks A, C, and D. We evaluated the change in the number of completed sessions with a Fishers-Exact test, and found the change to be statistically significant for the overall increase in completed sessions with HOOGLE+ ($p = .009$), and for tasks A ($p = .003$), and D ($p = .080$). While more users completed task C with HOOGLE+ than in the control setting, this change is not statistically significant ($p = .5$).

In task B there is virtually no association between the setting used and completions ($p = .715$), and the low completion rate seems to be more influenced by the difficulty of producing the typeclass constraint in the searched type.

Completion time. HOOGLE+ improved the average time to complete a task by 35 seconds. Average times are shown in Figure 1.12b.

Since tasks vary in components and difficulty, we also examine the data per-task. The improvement is preserved in tasks A, C, and D. We evaluated the change in time-to-complete with a Mann-Whitney U-test, and found the change statistically significant for tasks A ($p = .0003$) and C ($p = .051$), but neither the improvement in task D ($p = .354$) nor the 5 second increase in task B ($p = .460$) are statistically significant. The tool overall enjoys statistical significance over control ($p = .004$).

Additionally, we examined only the times to complete when the user did not time out, shown in Figure 1.12c. This allow us to take a closer look at how much help was HOOGLE+ when it does help. While the aggregate difference is smaller, a mere 15s improvement, we notice that in the individual tasks, differences are intensified. We also notice that for two tasks, B and D, the trend has reversed itself: users who were helped by HOOGLE+ completed task B, on average, a full minute faster, and task D almost a minute slower. Even still considering only those who completed their task, we do not find these differences statistically significant in task B ($p = .165$) or task D ($p = .386$). We observe similar significance for the remaining tasks (task A: $p = .0002$, task C: $p = .060$, overall: $p = .005$).

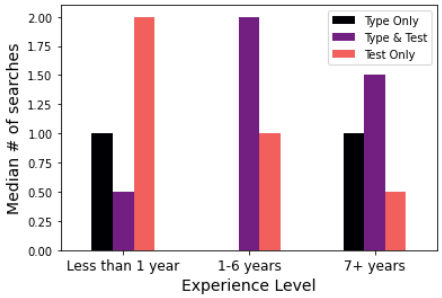


Figure 1.13: Median search modality across all four tasks, by experience level

We conjecture that task B required familiarity with typeclasses, so for those unfamiliar with the feature, HOOGLE+ could not help them; and, those with that knowledge could fly. Further, task D's expected solution may have been obvious to some and could easily write it out in control; yet, those in the treatment setting had to coax HOOGLE+ to generate the right candidate with enough examples.

Correctness. We logged the final solutions presented if users did not time out. Between both control and treatment, across 120 recorded tasks, and 73 total completions, only one participant concluded with an incorrect solution, for task A, using HOOGLE+. While this falls entirely within the margin of error, we do discuss the particulars of the session further in the next subsection.

Overall, we see that HOOGLE+ greatly improves completion rates over the control setting, as well as a modestly improving the time to result. Therefore **we answer RQ1 in the affirmative.**

RQ2: How do functional programmers express their intent in the synthesizer?

We logged user searches made in the course of the experiment, and analyzed the style of HOOGLE+ searches users made. HOOGLE+ permits 3 kinds of searches: (1) *type-only* search, leaving the test part of the specification empty, (2) *test-only* search, then using a type that HOOGLE+ suggested, and (3) *type-and-test* search. Users made a total of 115 searches across all HOOGLE+ sessions, with users making on average a little fewer than 2 searches per task. Only

22 searches were type-only, leaving 93 searches involving at least one test.

The style of search varied greater by experience level than by task. The breakdown of these searches is shown in Figure 1.13. Experts relied on tests the least, making a median of 0.5 test-only searches across all tasks, while inexperienced Haskell users made a median of 2 test-only queries. Despite our pre-study survey discovering that searching HOOGLE by type was the most popular way to query for a component, searching by type-only in our synthesis setting was uniformly the least popular mode.

Test Provenance. Tests were an important part of how participants made their searches. We note where these tests came from. Task descriptions included one ready-made test. Of the tests used in searches, 46 were directly from the task; 63 were original to the participant (though some were closely based on the task or what was on screen); only 2 tests came from examples provided by HOOGLE+.

To answer RQ2: across the board, users searched by type *the least* during their HOOGLE+ sessions. While beginners preferred test-only searches significantly, **tests were overwhelmingly part of user searches.** Additionally, users have a strong preference for **providing their own tests**.

RQ3: How useful are HOOGLE+ features in interpreting results?

We asked users to fill out a questionnaire after completing the tasks to assess what parts of HOOGLE+ they used and what they found most helpful. The ratings of HOOGLE+ features by users who used them (i.e., did not mark "did not use" in the survey) appear in Figure 1.14.

In general, users found HOOGLE+ features to be helpful or very helpful. The only features rated very unhelpful by any user were the documentation available when hovering on a component and the type-only search, which, as seen earlier, was also the least used of all search options. The users dissatisfied with the documentation liked the idea but indicated they wanted a different experience around reading the documentation inline.

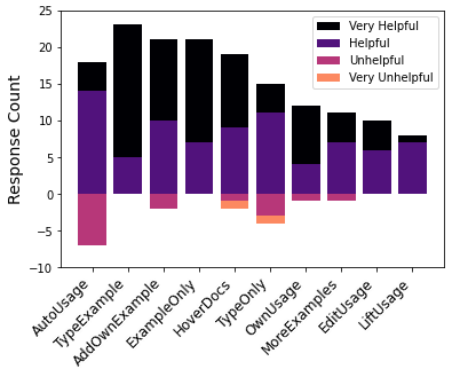


Figure 1.14: Breakdown of feature perception, ordered from most used to least used.

The less-used features of HOOGLE+, editing and lifting a usage, were used by participants who needed their functionality, so it is not surprising they also found them helpful. A non-negligible number of users found auto-generated examples unhelpful, which we will discuss in the next subsection.

To answer RQ3, with the exception of the auto-generated examples, **HOOGLE+ features are useful to users in interpreting results**.

1.7.3 Discussion

Overall effect on aid. Overall, the effect of HOOGLE+ on user performance was very encouraging, and feedback from participants was positive. In fact, one user said they felt they didn't really solve the task–the tool did–and that it felt like cheating at programming!

The four different tasks tested in our experiment are varied and stress different parts of a participant's Haskell knowledge. Task B required knowing or picking a typeclass, and task A involved lesser-known components in the `Data.Maybe` library. The control setting required the participants to come up with intermediate types for function compositions, in order to search the individual functions by type on HOOGLE. In our experience, about half of the participants did not know what the intermediate types should be in a solution *a priori*.

We observed that the task either immediately made sense to participants or they struggled

with it. In the data, we see a clear bimodal performance curve in both control and treatment, between those who "got it" and those who timed out or almost timed out. Task D is the most extreme example of this, causing the time to completion of those who finished the task in the control setting to be extremely fast (e.g., one participant solved the task in under a minute, saying they encountered a similar problem in their work). Still, more participants could solve task D with HOOGLE+ than without, showing that its value is in the cases that don't immediately click.

In tasks A and C the effect of using HOOGLE+ was most significant, both in completion and in the change in times. We believe that these two tasks were particularly hard to break down into intermediate types and component-searches and this played to our tool's strengths. A basic assumption of human-in-the-loop synthesis is often that the programmer is capable of helping the synthesizer break down the task. It is possible that in a functional setting, this assumption does not hold.

Barriers. We asked users about barriers to solving their tasks after both the control portion and the treatment portion of the study.

After the control setting, several users expressed feeling daunted by the task of coming up with the right intermediate types and searching for the right function that contains what they need. This ties in to the significantly slower control times in the tasks A and C that require uncommon components and complex, higher-order types, respectively.

Additionally, HOOGLE users had frustrations about the tool itself. Results often contain cruft from domain specific libraries that are usually not the function or direction intended. One user explicitly named as a barrier the need to browse a large set of results on a simple search. Several users mentioned vaguely remembering the necessary function, and having to search HOOGLE to recall the order of arguments or the precise name of the function, but HOOGLE doesn't permit searches by documentation.

The most frequent barrier to HOOGLE+ users were slow synthesis times. Specifically, the lack of indication if a search that was taking long would yield results or wait and then return

nothing. Additionally, users expressed the need for messaging suggesting actions to the user when no results came up.

Several users mentioned difficulty in understanding what the candidate functions were actually doing, because any example provided was only shown as an end-to-end execution. One participant suggested drilling down into a candidate's execution on an example would help.

These point to experience and design improvements that are needed for HOOGLE+ to become an effective production tool, but are not insurmountable.

Search Style. In our observations, we found that participants would fall back to example-only searches when they were at a loss for the right type (mostly with the most novice participants), or when they wanted to let the tool do more work for them. One participant made the observation that, "the point of a tool is to take the thinking out".

Task B is a particularly interesting case: only two participants searched with types-alone, the fewest of any task. Perhaps most users could tell the function's type signature `Eq a => [a] → [a]` is very underspecified– that it says very little about what should happen to inputs– and so included at least one test with their search. This highlights the occasional shortcomings of types as specifications, ones that are mitigated by allowing tests in the search specification.

Auto-generated Examples. As shown in Figure 1.14, the auto-generated usage examples for candidate programs were the HOOGLE+ feature users were least satisfied with. We observed that this stemmed mainly from user expectation of usage examples did not entirely aligning with the criteria for example generation (subsection 1.5.2). Specifically, users did not need differentiation between the candidates as much as they wanted usages to explore the functionality of the current program they are investigating.

Users who did try to understand the candidates via the generated examples wished for a greater diversity of examples. Those who did ask for more examples tended to ask for *many* more examples, 6.7 more, on average, with some clicking the button up to 17 times (between both

tasks). This shows that these users were hoping for the system to help them better understand their candidates.

This is perhaps best illustrated by the only incorrect result out of 60 HOOGLE+ tasks performed. The user, a Haskell novice, made use of test-only searches but selected a type too specific for the task. They then selected a candidate that appeared to fit the task description but would crash on inputs they never tested. The user *did investigate* the candidate by asking for more examples and editing existing ones; however, the user did not attempt any complex inputs. This user's experience demonstrates room for improvement in our example generation—better aligning its goals with the needs of users, and producing a greater variety of examples.

1.7.4 Threats to Validity

We selected tasks and components to operate over several common, built-in libraries. Most participants were familiar with many functions but had to limit themselves to the subset we permitted in the control setting. This introduced "unintentional complexity" as one expert user aptly put. We attempted to mitigate this with a training task in the control setting to familiarize users with our restrictions.

We gave participants only 8 minutes to complete each task. This short time limit is lab induced, and some participants reported a sort of test-anxiety that may have affected their performance. Anecdotally, many participants were close to completing the task in both control and experiment after timing out. Since the data is right-censored, times over eight minutes are only known to be over eight minutes, which may make generalizing the results for more complex tasks incorrect.

1.8 Related Work

Component-based Synthesis. Modern IDEs support code-completion based on matching common prefixes of names (*e.g.* completing `Str` into `String`), or by using the context to narrow the candidates to well-typed completions [90]. Type-based search engines like Hoogle [77] generalize the above to find type isomorphisms [28] *i.e. single* components whose signature match the query. In contrast, our goal is to find *combinations* of components that implement some higher-level task. When the task is specified as a type, the problem of search reduces to that of *type inhabitation*, *i.e.* finding terms that inhabit a given query type [116]. One approach to type inhabitation is proof search [5, 80, 50], which can be difficult to scale up to large component libraries. PROSPECTOR [73] introduces a scalable graph-based inhabitation algorithm where the components are unary functions, SYPET [33] uses Petri-nets to generalize graph-based methods to multiple argument functions, and TYGAR [46] shows how to further extend SYPET's search to polymorphic components using the idea of *succinct* type-abstractions introduced by INSYNTH [47]. However, all of these require type-based queries which can be problematic for non-experts, and do not consider the question of end-to-end usability.

User Interaction in Program Synthesis. Although program synthesis is supposed to serve a user, few papers focus on the user's role in the synthesis loop. [65] and [87] highlight two models of iterative synthesis, the first driven by the synthesizer and the second by the user. Our work is in a different setting: API discovery for functional languages.

Several domain-specific synthesizers [17, 30, 21] give end-users and data scientists access to synthesis to automate some of their work. These tools were evaluated against users' alternative workflow, but their users are not programmers, and the synthesis domain is far from general. Unlike these, HOOGLE+ is a tool for (functional) programmers and allows users to search for general Haskell code.

Filtering and ranking synthesis results. Ranking and returning multiple results are two

common approach to handing ambiguous specifications in program synthesis; the two often—but not always—go hand-in-hand. The FLASHX tool family [42, 96] uses a ranking function to select a single, most likely program from programs that satisfy all user-provided examples, exploring both hand-crafted [42] and learned [106] ranking functions. Recent work on synthesizing lenses [76] proposed a novel approach to semantic ranking based on information theory. Unlike PBE tools that use ranking to select a single result, code completion tools [47, 97] typically present a ranked list of results to the user, and most commonly rely on learned statistical models and syntactic features. Like these tolls, HOOGLE+ offers the users several ranked candidates, both of synthesis results and of inferred types.

Synthesizers also need to filter their results to discard irrelevant programs. SYPET [33] uses Petri-nets to only return programs that use all available arguments. HOOGLE+ extends this filtering: it filters TYGAR results after they are constructed, and uses more extensive criteria.

Test input generation. The extensive literature on automating testing focuses on finding bugs in manually written code. Our key observation is that these ideas in general, and property-based testing in particular, can be re-purposed for example-based elimination and comprehension in program synthesis. HOOGLE+ uses the SMALLCHECK library [101] to filter its candidate program list and to provide examples to demonstrate the semantics of synthesized programs.

Inferring Types from Examples. A key innovation of HOOGLE+ is to allow users to specify their queries via tests that are then translated into types, enabling efficient search. Prior work on the problem of inferring types from tests has a very different context: inferring type annotations for dynamically typed languages. E.g., [22] infer types from run-time logs, [4] instrument Ruby programs to track how each variable is *used* to then build a constraint system that is solved to infer method types, and [10] show how to generalize execution-based guided type-recovery to handle ad-hoc recursive datatypes as found in Clojure programs. All these differ from our approach in several ways. First, the different setting: when discovering type annotations, they have program execution traces to help guide type inference. Second, all infer

monomorphic types, while our goal is to infer polymorphic signatures greatly narrowing the synthesizer search space.

1.9 Conclusion

In this work we presented HOOGLE+, a component-based synthesizer for Haskell that focuses on end-to-end usability of program synthesis. HOOGLE+ extends a core type-driven synthesis engine TYGAR in three major ways. First, we present a novel mechanism to infer likely polymorphic type signatures from tests, which helps beginners, who are not yet fully comfortable with the Haskell type system. Second, we show how to leverage property-based testing to eliminate meaningless and repetitive synthesis results, without asking the user for additional input. Finally, we again rely on property-based testing to automatically generate examples that demonstrate the behavior of synthesized programs.

To evaluate the usefulness of HOOGLE+ relative to a traditional code search workflow, we conducted a user study with 30 participants, comparing their performance on solving simple programming tasks with the HOOGLE search engine *vs.* HOOGLE+. We find that users equipped with HOOGLE+ perform their search tasks faster are able to complete 50% more tasks.

The authors would like to thank the anonymous reviewers for their feedback on the draft of this paper. This work was supported by the National Science Foundation under Grants No. 1943623 and 1911149.

Chapter 1, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA. Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Nadia Polikarpova. ACM 2020. The author was a principal author and investigator on this work.

Chapter 2

Program Snippet Recognition

2.1 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 [**?**]. Zhang et al. allow a user to request examples to test corner cases in regular expression synthesis [133]. 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.

2.2 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 Kelly asks her IDE's synthesizer to complete the snippet given the type and it provides 5 possible results, as mocked-up in Figure 2.1. Kelly wants to quickly figure out if any

```haskell
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 2.1: 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.

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 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.

2.3 Related Work

Program comprehension typically focuses on how a programmer understands an entire software system or part of it [72]. Comprehension research has focused on how a program-

ming paradigm affects understanding [102]; how developers understand patches [115]; 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 (like in Chapter 1). Peleg and Polikarpova's BESTER synthesis engine provides multiple snippets and shows the user which examples a given snippet successfully passes [86]. 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 [29]. WREX provides feedback, helping a user see if the synthesized code accomplishes their data-wrangling task; however, it is limited to Flashfill-like domains [43]. In more the tightly controlled domain of regular expressions, Zhang et al. help a user select a synthesized regular expression matching their intent [133]. 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.

2.4 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 recog-

nition 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 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?**

2.4.1 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.

2.4.2 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

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

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 2.2: The functions participants were asked to implement using snippets provided to them.

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 2.2). `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.

2.5 Observations

We observed our participants and gathered commonly used techniques.

2.5.1 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 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.

2.5.2 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 [133], 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.

Acknowledgements. Chapter 2, in part, is a reprint of the material as it appears in the 12th Annual Workshop at the Intersection of PL and HCI. Michael B. James and Nadia Polikarpova. PLATEAU 2021. The author was a principal author and investigator on this work.

Chapter 3

Grounded Understanding of LLMs for Programming

3.1 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 [98, 75, 35, 79] and the machine learning [59, 128, 45] communities. Thanks to recent breakthroughs in large language models (LLMs) [120, 68] this dream finally seems within reach. OpenAI's Codex model [20], 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 [68] 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 [38].

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 [117]; although their work makes several interesting observations about human behavior (which we discuss in more detail in section 3.7), 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) [39], a qualitative research technique that has a long history in social sciences, and has recently been adopted to study phenomena in software engineering [110] and programming languages [70]. 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 section 3.3.

Summary of findings. The main thesis of our theory (section 3.4) 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 [14, 74], popularized by Daniel Kahneman's "Thinking, Fast and Slow" [58]. 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.

Section 3.5 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 (section 3.6). 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.

3.2 Copilot-Assisted Programming, by Example

Copilot is a programming assistant released by Github in June 2021 [38], and since integrated into several development environments, including Visual Studio Code, JetBrains and Neovim. Copilot is powered by the OpenAI Codex family of models [20], which are derived by fine-tuning GPT-3 [11] on publicly available Github repositories.

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

3.2.1 Copilot as Intelligent Auto-Completion

Axel, a confident Python programmer, is solving an Advent of Code [125] 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` (Figure 3.1). 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. Figure 3.1 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

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

preceding the cursor, which can include both code and natural language comments, as is the case in our example.

Because the suggestion in Figure 3.1 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 [14], which is characterized by quick, automatic, and heuristic decisions.

3.2.2 Copilot as an Exploration Tool

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 Figure 3.2. Moreover, since she wants to explore multiple options, she presses <ctrl>

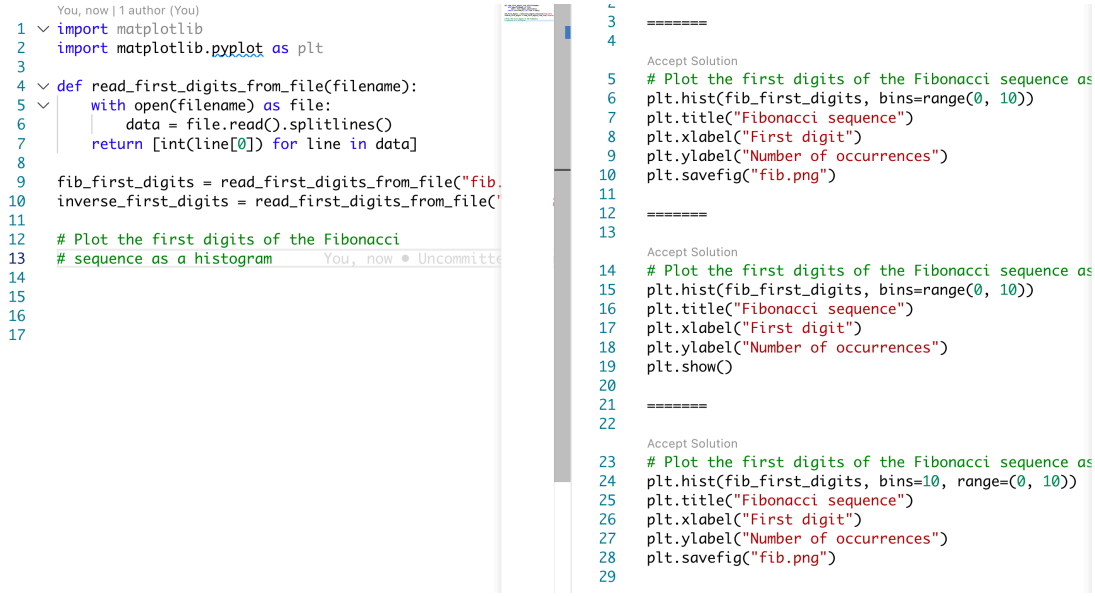


Figure 3.2: The user writes an explicit comment prompt (lines 12–13 on the left) and invokes Copilot's multi-suggestion pane by pressing `<ctrl> + <enter>`. The pane, shown on the right, displays up to 10 unique suggestions, which reflect slightly different ways to make a histogram with `matplotlib`.

`+ <enter>` to bring up the *multi-suggestion pane*, which displays up to 10 unique suggestions in a separate pane (shown on the right of Figure 3.2). 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.

Table 3.1: 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

3.3 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. Table 3.1 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 Table 3.1: Never, Occasional, Regular, and Professional). We screened out participants who had never used the target language. We choose a qualitative self-assignment of experience as other common met-

[1]Participant did not have time to attempt Python section

rics, 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 [39]. 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 [111], commonly called *Straussian Grounded Theory* [110]. 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 section 3.4.1). Once in this acceleration mode, programmers would validate a suggestion by a kind of visual "pattern matching" (details in section 3.4.1). 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 Table 3.2. The full task templates we provided to participants are available in our replication package [7].

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 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 subsection 3.2.2. 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 subsection 3.2.1). 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

[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 3.2: 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

we collected rounded out our hypotheses about validation (section 3.4.1, section 3.4.2).

3.4 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 subsection 3.2.1 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 subsection 3.2.2, 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* (subsection 3.4.1) and *exploration* (subsection 3.4.2). 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.* section 3.4.1) is a hypothesis deriving from our top-level bimodal theory. Each named paragraph heading is an aspect of that hypothesis.

3.4.1 Acceleration

Acceleration is characterized by the programmer being "in the zone" and actively "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 section 3.5.

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 [89], 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.”

3.4.2 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 (section 3.4.1); 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 [35].

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. Af-

ter 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 *execu-*

tion (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 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.

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

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 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.

3.5 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.

3.5.1 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. Figure 3.3 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:

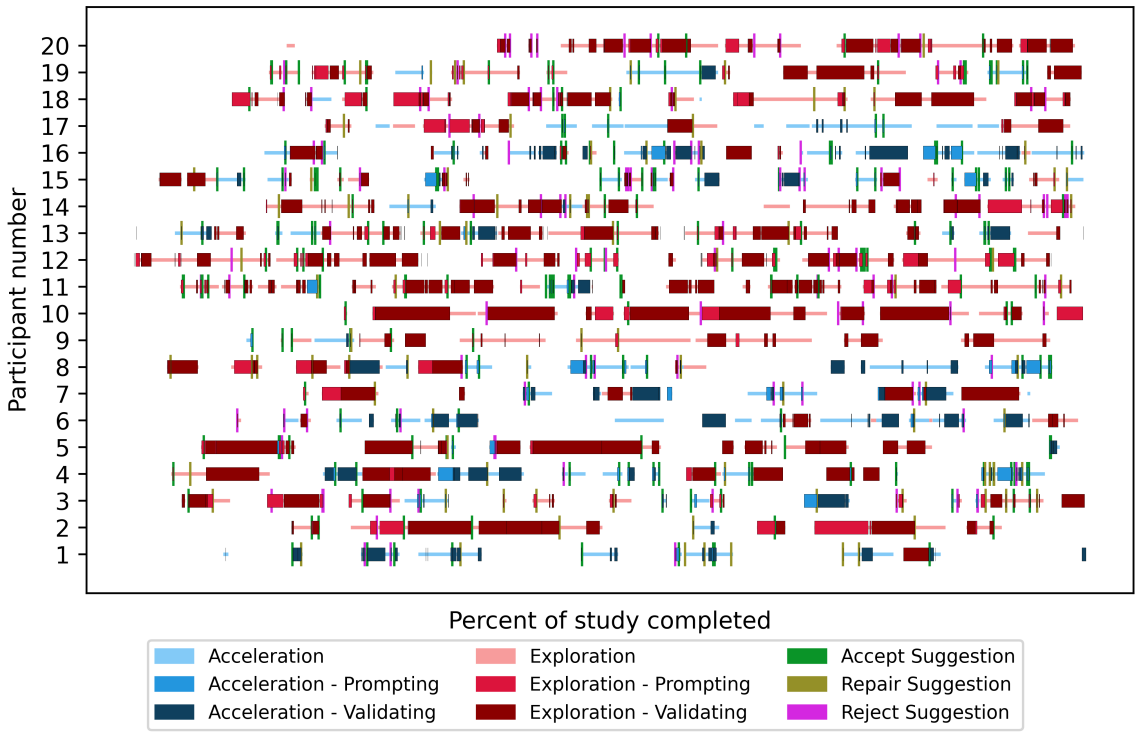
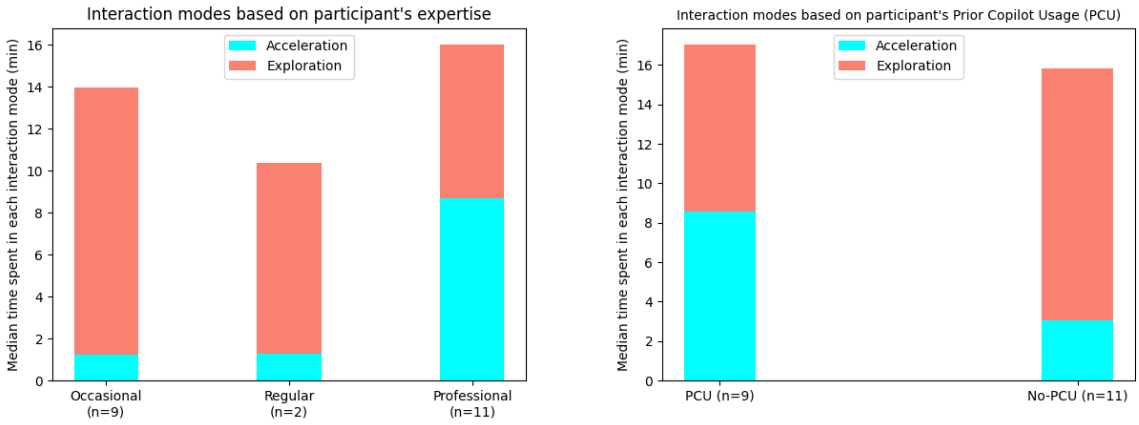


Figure 3.3: Timeline of observed activities in each interaction mode for the 20 study participants. The qualitative codes include different prompting strategies, validation strategies and outcomes of Copilot's suggestions (accept, reject or repair)



(a) Grouped by language expertise. (b) Grouped by prior Copilot usage.

Figure 3.4: Median time spent in acceleration vs exploration mode for different participant groups.

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

Language expertise. Figure 3.4a 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

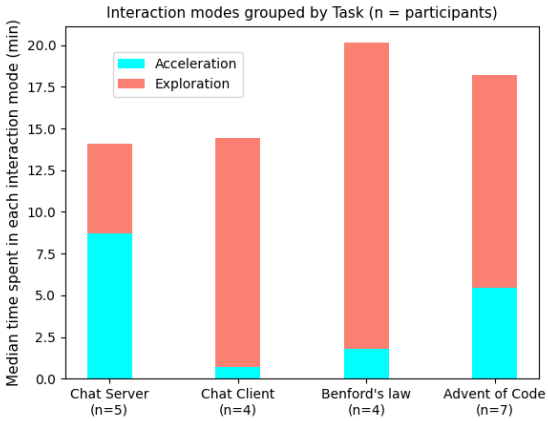


Figure 3.5: Median time spent in acceleration vs exploration mode, grouped by task.

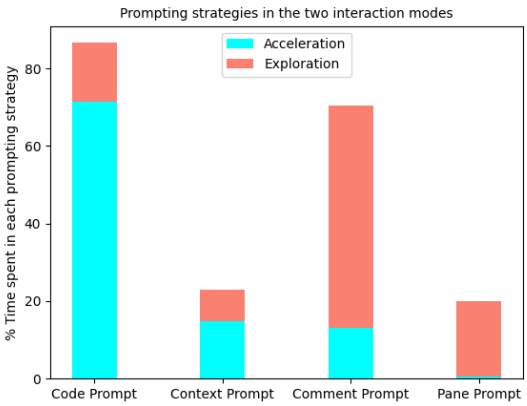


Figure 3.6: Prevalence of prompting strategy as percentage of total prompting time.

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 Figure 3.4b 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 section 3.4.2, where novice users have higher expectations of Copilot's ability to solve high-level exploratory tasks.

Nature of Task. Figure 3.5 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.

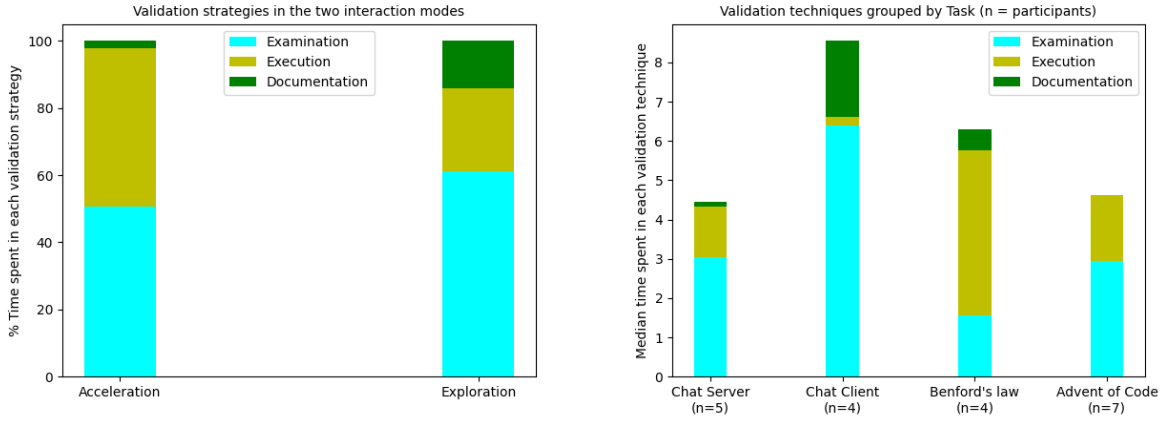
Programming Language. We did not identify any noticeable differences in either total

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 (subsection 3.2.1) and intentional prompting (subsection 3.2.2). 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 (section 3.4.2) or invoking the multi-suggestions pane by pressing `<ctrl> + <enter>` (section 3.4.2).

Figure 3.6 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 it provides a rich body of suggestions for participants to explore from.



(a) Aggregate time, split by interaction mode. (b) Median time, grouped by task.

Figure 3.7: Time spent in different validation strategies.

Validation Strategies across Interaction Modes and Tasks

Recall that section 3.4.2 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.

Figure 3.7a 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 Figure 3.7b. 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.

3.5.2 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 subsection 3.5.1. 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).

3.6 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.

3.6.1 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 [88]. 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.

3.6.2 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 section 3.4.1, 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 section 3.4.2, 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 [40] supports exploring a space of student solutions to a programming assignment.

Suggestions with holes. Recall from section 3.4.2, that when programmers modify suggestions, 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 [45] 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 [70] 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 [24] to run automatically on Copilot suggestions. P14 wished they had *projection boxes* [66], 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.

3.7 Related Work

Usability of Copilot. The closest to our work is the study by [117], which also evaluates Copilot. The main differences are: (1) their study is on stand-alone tasks, whereas ours includes tasks that require contributing to an existing codebase; (2) their study is comparative and focuses on the rate and time of task completion with and without Copilot's help; (3) their study only used Python, whereas we used several programming languages. 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, [117] 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, [57] 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, [129] investigated the usefulness of an NL-to-code plugin previously developed by the same authors [128]. 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, [126] 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).

[104] 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 [112] and Kite [61], 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 [54], regular expressions [134, 132], web scraping [18], or data querying, wrangling, and visualization [31, 123, 137]. 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 [85] and Snippy [35, 34] 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, [35] 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 section 3.4.2), although it appears to be less prominent in LLM-based tools, since their performance degrades more gradually with the complexity of the task.

[55] 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 (section 3.4.2), 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 [15]. [110] 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: [70] study how statically-typed functional programmers write code, and deliver a set of guidelines meant for functional language tool-builders.

3.8 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!

The authors would like to thank Devon Rifkin from GitHub for his assistance with getting Copilot access for our study participants. This work was supported by NSF grants 2107397 and 1955457.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 7, Issue OOPSLA1. Shraddha Barke, Michael B. James, Nadia Polikarpova. ACM 2023. The author was a principal author and investigator on this work.

Chapter 4

Validating AI-Generated Code with Live Programming

4.1 Introduction

Recent advances in large language models have given rise to AI-powered code suggestion tools like GitHub Copilot [38], Amazon CodeWhisperer [3], and ChatGPT [81]. These *AI programming assistants* are changing the face of software development, automating many of the traditional programming tasks, but at the same time introducing *new tasks* into the developer's workflow—such as prompting the assistant and reviewing its suggestions [8, 78]. Development environments have some catching up to do in order to provide adequate tool support for these new tasks.

In this paper, we focus on the task of *validating* AI-generated code, *i.e.*, deciding whether it matches the programmer's intent. Recent studies show that validation is a bottleneck for AI-assisted programming: according to [78], it is the *single most prevalent activity* when using AI code assistants, and other studies [117, 69, 124, 9] report programmers having trouble evaluating the correctness of AI-generated code. Faced with difficulties in validation, programmers tend

to either *under-rely* on the assistant—*i.e.*, lose trust in it—or to *over-rely*—*i.e.*, blindly accept its suggestions [126, 100, 119, 113]; the former can cause them to abandon the assistant altogether [8], while the latter can introduce bugs and security vulnerabilities [91]. These findings motivate the need for better validation support in AI-assisted programming environments.

This paper investigates the use of *Live Programming* (LP) [48, 121, 114] as a way to support the validation of AI-generated code. LP environments, such as Projection Boxes [67], visualize runtime values of a program in real-time without any extra effort on the part of the programmer. We hypothesize that these environments are a good fit for the validation task, since LP has been shown to encourage more frequent testing [12] and facilitate bug finding [136] and program comprehension [27, 26, 13]. On the other hand, validation of AI-generated code is a new and unexplored domain in program comprehension, which comes with its unique challenges, such as multiple AI suggestions for the programmer to choose from, and frequent context switches between prompting, validation, and code authoring [78], which cause additional cognitive load [124]. Hence, the application of LP to the validation setting warrants a separate investigation.

To this end, we constructed a Python environment that combines an existing LP environment [67] with an AI assistant similar to Copilot's multi-suggestion pane. Using this environment, we conducted a between-subjects experiment ($N = 17$) to evaluate how the availability of LP affects users' effectiveness and cognitive load in validating AI suggestions. Our study shows that Live Programming facilitates validation through *lowering the cost of inspecting runtime values*; as a result, participants were more successful in evaluating the correctness of AI suggestions and experienced lower cognitive load in certain types of tasks.

4.2 Related Work

Validation of AI-Generated Code A rapidly growing body of work analyzes how users interact with AI programming assistants. Studies show that programmers spend a significant proportion of their time validating AI suggestions [8, 78, 9]. Moreover, a large-scale survey [69] indicates that 23% of their respondents *have trouble evaluating correctness of generated code*, which echoes the findings of lab studies [117, 8] and a need-finding study [124], where participants report difficulties understanding AI suggestions and express a desire for better validation support. [8] and [69] find that programmers use an array of validation strategies, and the prevalence of each strategy is *closely related to its time cost*. Specifically, despite the help of execution techniques built into the IDE for validating AI suggestions [113], execution is used less often than quick manual inspection or type checking because it is more time-consuming [8, 69] and interrupts programmers' workflows [124]. The lack of validation support designed for AI-assisted programming, as [124] identify, leads to a higher cognitive load in reviewing suggestions. The high cost of validating AI suggestions, according to some studies [126, 100, 119], can lead to both *under-reliance*—lack of trust—and *over-reliance*—uncritically accepting wrong code—on the part of the programmer.

Comparatively fewer existing papers explore interface designs to support validation of AI-generated code: [100] investigates a conversational assistant that allows programmers to ask questions about the code, while [118] targets over-reliance by highlighting parts of generated code that might need human intervention; our work is complementary to these efforts in that it focuses on facilitating validation by execution.

Validation in Program Synthesis Another line of related work concerns the validation of code generated by *search-based* (non-AI-powered) program synthesizers. Several synthesizers help users validate generated code by proactively displaying its outputs [31, 134, 54] and intermediate trace values [85], although none of them use a full-fledged LP environment. The only system

we are aware of that combines LP and program synthesis is SNIPPY [36], but it uses LP to help the user specify their intent rather than validate synthesized code.

Live Programming Live Programming (LP) provides immediate feedback on code edits, often in the form of visualizations of the runtime state [48, 121, 114]. Some quantitative studies find that programmers with LP find more bugs [136], fix bugs faster [63], and test a program more often [12]. Others find no effect in knowledge gain [52] or efficiency in code understanding [13]. Still, qualitative evidence points to the helpfulness of LP for program comprehension [27, 26, 13] and debugging [60, 52]. In contrast to these studies, which evaluate the effectiveness of LP for comprehending and debugging *human-written* code, our work investigates its effectiveness for validating *AI-generated* code, a setting that comes with a number of previously unexplored challenges [78, 124].

4.3 LEAP: the Environment Used in the Study

To study how Live Programming affects the validation of AI-generated code, we implemented LEAP (**L**ive **E**xploration of **A**I-Generated **P**rograms), a Python environment that combines an AI assistant with LP. This section demonstrates LEAP via a usage example and discusses its implementation.

Example Usage Naomi, a biologist, is analyzing some genome sequencing data using Python. As part of her analysis, she needs to find the most common bigram (*i.e.*, two-letter sequence) in a DNA strand.[1] To this end, she creates a function `dominant_bigram` (line 3 in Figure 4.1); she has a general idea of what this function might look like, but she decides to use LEAP, an AI assistant, to help translate her idea into code.

[1]This is one of the programming tasks from our user study, and each of Naomi's interactions with LEAP has been observed in some of our participants.

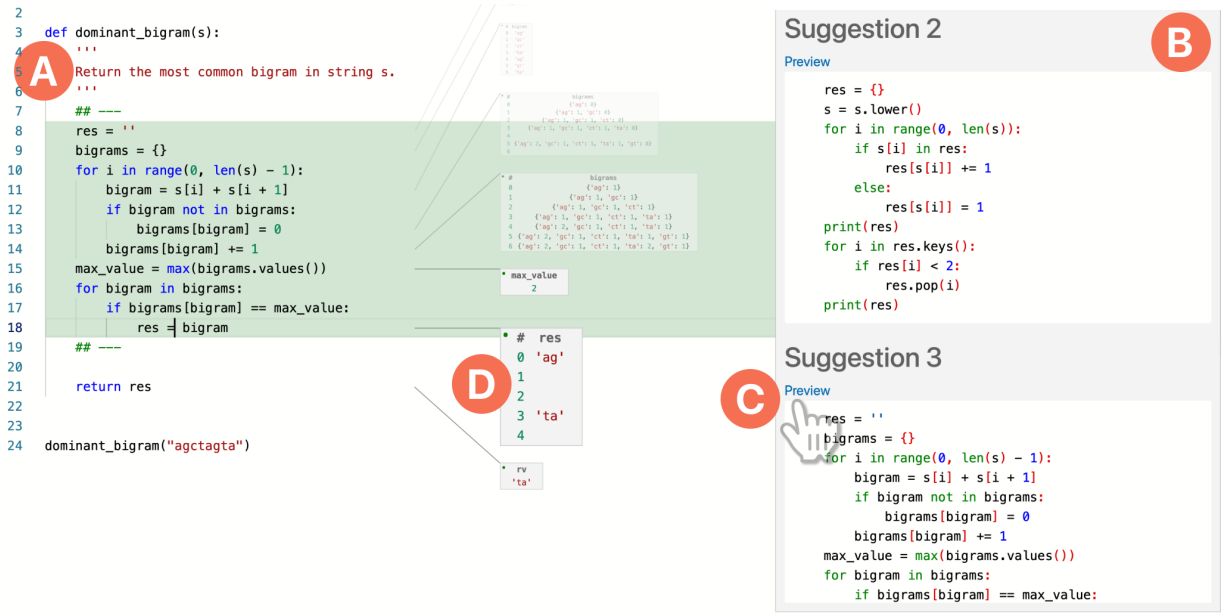


Figure 4.1: LEAP is a Python environment that enables validating AI-generated code suggestions via Live Programming.
`(A)` Users prompt the AI assistant via comments and/or code context. `(B)` The Suggestion Panel shows the AI-generated suggestions. `(C)` Pressing a `Preview` button inserts the suggestion into the editor. `(D)` Users can inspect the runtime behavior of the suggestion in Projection Boxes [67], which are updated continuously as the user edits the code.

(A) Naomi adds a docstring (line 5), which conveys her intent in natural language, and a test case (line 24), which will help her validate the code. With the cursor positioned at line 7, she presses `Ctrl` and `Enter` to ask for suggestions.

(B) Within seconds, a panel opens on the right containing five code suggestions; Naomi quickly skims through all of them. The overall shape of Suggestion 3 looks most similar to what she has in mind: it first collects the counts of all bigrams into a dictionary, and then iterates through the dictionary to pick a bigram with the maximum count.

(C) Naomi decides to try this suggestion, pressing its `Preview` button; LEAP inserts the code into the editor and highlights it (lines 8-18).

(D) As soon as the suggestion is inserted, Projection Boxes [67] appear, showing runtime information at each line in the program. Inspecting intermediate values helps Naomi understand what the code is doing step by step. When she gets to line 18, she realizes that the dictionary actually has *two* dominant bigrams with the same count, and the code returns *the last one*. She realizes this is not what she wants:

instead, she wants to select the dominant bigram that comes first alphabetically (`ag` in this case).

One option Naomi has is to try other suggestions. She clicks on the `Preview` button for Suggestion 2; LEAP then inserts Suggestion 2 into the editor, in place of the prior suggestion, and the Projection Boxes update instantly to show its behavior. Naomi immediately notices that Suggestion 2 throws an exception inside the second loop, so she abandons it and goes back to Suggestion 3, which got her closer to her goal.

To fix Suggestion 3, Naomi realizes that she can accumulate all dominant bigrams in a list, sort the list, and then return the first element. She does not remember the exact Python syntax for sorting a list, so she tries different variations—including `l = l.sort`, `l = l.sort()`, `l = sort(l)`, `l = l.sorted()`, and so on. Fortunately, LEAP's support for LP allows Naomi to get immediate feedback on the behavior of each edit, so she iterates quickly to find one of the correct options: `l = sorted(l)`. Note that Naomi's workflow for using Suggestion 3—validation, finding bugs, and fixing bugs—relies on full LP support, and would not work in traditional environments like *computational notebooks*, which provide easy access to the final output of a snippet but not the intermediate values or immediate feedback on edits.

Implementation To generate code suggestions, LEAP uses the `text-davinci-003` model [82], the largest publicly available code-generating model at the time of our study. To support live display of runtime values (Figure 4.1 `(D)`), we built LEAP on top of Projection Boxes, a state-of-the-art LP environment for Python [67] capable of running in the browser. As the control condition for our study, we also created a version of LEAP, where Projection Boxes are disabled, and instead the user can run the code explicitly by clicking a `Run` button and see the output in a terminal-like Output Panel.

4.4 User Study

We conducted a between-subjects study to answer the following research questions:

RQ1) How does Live Programming affect over- and under-reliance in validating AI-generated code?

RQ2) How does Live Programming affect users' validation strategies?

RQ3) How does Live Programming affect the cognitive load of validating AI-generated code?

Tasks Our study incorporates two categories of programming tasks, *Fixed-Prompt* and *Open-Prompt* tasks.

In *Fixed-Prompt tasks*, we provide participants with a *fixed set* of five AI suggestions that are intended to solve the entire problem. We curated the suggestions by querying Copilot [38] and LEAP with slight variations of the prompt. Fixed-Prompt tasks isolate the effects of Live Programming on validation behavior by controlling for the quality of suggestions. We created two Fixed-Prompt tasks, each with five suggestions: (T1) *Bigram*: Find the most frequent bigram in a given string, resolving ties alphabetically (same task in Section 4.3); (T2) *Pandas*: Given a `pandas` data frame with data on dogs of three size categories (small, medium, and large), compute various statistics, imputing missing values with the mean of the appropriate category. These tasks represent two distinct styles: Bigram is a purely algorithmic task, while Pandas focuses on using a complex API. Pandas has two correct AI suggestions (out of five) while Bigram has none, a realistic scenario that programmers encounter with imperfect models.

In *Open-Prompt tasks*, participants are free to invoke the AI assistant however they want. This task design is less controlled than Fixed-Prompt, but more realistic, thus increasing ecological validity. We used two Open-Prompt tasks: (T3) *String Rewriting*: parse a set of string transformation rules and apply them five times to a string; (T4) *Box Plot*: given a `pandas` data frame containing 10 experiment data records, create a `matplotlib` box plot of time values for each group, combined with a color-coded scatter plot. Both tasks are more complex than the Fixed-Prompt tasks, and could not be solved with a single interaction with the AI assistant.

Participants and Groups We recruited 17 participants; 5 self-identified as women, 10 as men, and 2 chose not to disclose. 6 were undergraduate students, 9 graduate students, and 2 professional engineers. Participants self-reported experience levels with Python and AI assistants: 2 participants used Python 'occasionally', 8 'regularly', and 7 'almost every day'; 7 participants declared they had 'never' used AI assistants, and 8 used such tools 'occasionally'.

There were two experimental groups: "LP" participants used LEAP with Projection Boxes, as

described in Figure 4.1; "No-LP" participants used LEAP *without* Projection Boxes, instead executing programs in a terminal-like Output Panel. Participants completed both Fixed-Prompt tasks and one Open-Prompt task. We used block randomization [32] to assign participants to groups while evenly distributing across task order and selection and balancing experience with Python and AI assistants across groups. The LP group had 8 participants, and No-LP had 9.

Procedure and Data We conducted the study over Zoom as each participant used LEAP in their web browser. Each session was recorded and included two Fixed-Prompt tasks (10 minutes per task), two post-task surveys, one Open-Prompt task (untimed), one post-study survey, and a semi-structured interview. A replication package[2] shows the full details of our procedure, tasks, and data collection.

For *quantitative* analysis, we performed closed-coding on video recordings of study sessions to determine each participant's *subjective* assessment of their success on the task; we matched this data against the *objective* correctness of their final code to establish whether they succeeded in accurately validating AI suggestions. We also measured task duration—proportion of time Suggestion Panel (Figure 4.1`(B)`) was in focus—and participants' cognitive load (via five NASA Task Load Index (TLX) questions [49]). We used Mann-Whitney U tests to assess all differences except for validation success, which we analyzed via Fisher's exact tests.

In addition, we collected *qualitative* data from both Fixed-Prompt and Open-Prompt tasks. We noted validation-related behavior and quotes, which we discussed in memoing meetings [16] after the study. Through reflexive interpretation, we used category analysis [131] to assemble the qualitative data into groups. We then revisited notes and recordings to iteratively construct high-level categories.

4.5 Results

4.5.1 RQ1: Over- and Under-reliance on AI Assistants

To investigate if Live Programming affects over- and under-reliance, we measured whether participants successfully validated the AI suggestions in the Fixed-Prompt tasks, as described below. We

[2] `http://bit.ly/leap-study-package`

also compared task completion times and participants' confidence in their solutions (collected through post-task surveys). However, neither result was significantly different between the two groups, so we do not discuss them below.[3]

We found six instances of unsuccessful validation, all from the No-LP group. As described in Section 4.4, we compared subjective and objective assessments of code correctness on the two Fixed-Prompt tasks, which resulted in four outcomes: (1) *Complete and Accurate*, where the participant submitted a correct solution within the task time limit, (2) *Complete and Inaccurate*, where the participant submitted an incorrect solution without recognizing the error, (3) *Timeout after Validation*, where the participant formed an accurate understanding of the correctness of the suggestions but reached the time limit before fixing the error in their chosen suggestion, and (4) *Timeout during Validation*, where the participant reached the time limit before they had finished validating the suggestions. We consider (1) and (3) to be instances of *successful validation*, (2) to be an instance of *over-reliance* on the AI suggestions, and (3) to be an instance of *under-reliance*, as the participant did not successfully validate the suggestions in the given time. As Figure 4.2 shows, we found three instances of over-reliance in the Bigram task and three instances of under-reliance in the Pandas task, *all from the No-LP group*, though the overall between-group difference was not significant ($p = .206$ for both tasks).

Participants with over-reliance did not inspect enough runtime behavior.

The three No-LP participants with over-reliance in Bigram (P5, P12, P15) made a similar mistake: they accepted one of the mostly-correct suggestions (similar to Suggestion 3 in Section 4.3) and failed to notice that ties were not resolved alphabetically. Among the three participants, P5 did not run their code at all. P12 and P15 both tested *only one* suggestion on the given input and failed to notice the presence of two bigrams of the same count (and the fact that other suggestions returned different results). In addition, P15 cited *"reading the comments on what it was doing"* as a key factor for choosing the suggestion they

[3] In median times, the LP group completed the Pandas task faster by 35 seconds ($p = .664, U = 31$). For Bigram, LP participants were slower by 3 minutes and 51 seconds ($p = .583, U = 42$), though this difference changes to *faster* by 10 seconds if we exclude those who solved the task incorrectly. For Pandas, both groups had the median ratings of confidence in correctness as "Agree" on seen inputs ($p = .784, U = 30$) and "Neutral" on unseen inputs ($p = .795, U = 33$). For Bigram, the LP group had the median rating of confidence in correctness on seen inputs as "Agree", while the No-LP group had "Strongly Agree" ($p = .097, U = 19.5$). As for confidence in correctness on unseen inputs, the median for the LP group was "Neutral", and that for the No-LP group was "Agree" ($p = .201, U = 22.5$).

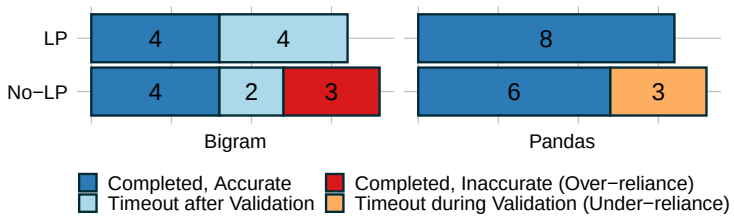


Figure 4.2: Success in validating AI suggestions across groups for Fixed-Prompt tasks. "Completed" means the participant submitted a solution they were satisfied with by the time limit, and "Timeout" means they did not. We deem the validation *successful* if a participant either submitted a solution that was correct (dark blue) or timed out when attempting to fix the correctly identified bugs in their chosen suggestion (light blue).

did. That suggestion began with a comment stating that it resolved ties alphabetically, but the following code did not do so.

Participants with under-reliance lacked affordances for inspecting runtime behavior. The three No-LP participants who under-relied on AI suggestions (P7, P9, P15) tried to use runtime values for validation but struggled with doing so. P9 previewed and ran multiple suggestions but did not add any `print` statements to the code, and so they could only see the output of one of the suggestions, which ended in a `print` statement. P15 ran all suggestions and did add a `print` statement to each to inspect the final return value, but the need to change the `print` statement and re-run each time made this process difficult, and they lost track of which suggestions they considered the most promising, saying *"I forgot which ones looked decent."* Finally, P7's strategy was to print the output of subexpressions from various suggestions in order to understand their behavior and combine them into a single solution, but this was time-consuming, so they did not finish.

4.5.2 RQ2: Validation Strategies

Our participants had access to two validation strategies: *examination* (reading the code) and *execution* (inspecting runtime values). The general pattern we observed was that participants first did some amount of examination inside the Suggestion Panel—ranging from a quick glance to thorough reading—and then proceeded to preview zero or more suggestions, performing further validation by execution inside the editor. To this end, No-LP participants in most tasks ran the code and added `print` statements for both final and intermediate values; LP participants in all tasks inspected both final and intermediate

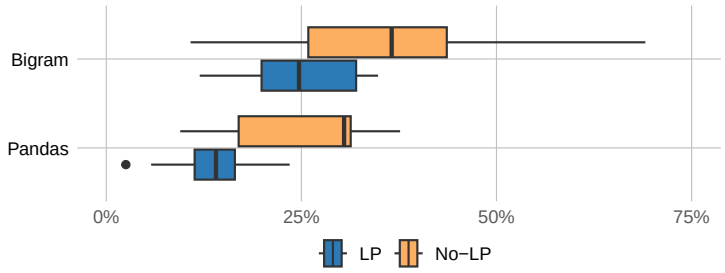


Figure 4.3: Percentage of time spent in the Suggestion Panel across the two groups for Fixed-Prompt tasks.

runtime values in Projection Boxes (by moving the cursor from line to line to bring different boxes into focus), and occasionally added `print` statements to see variables not shown by default. Below we discuss notable examples of validation behavior, as well as differences between the two groups and across tasks.

LP participants spent less time reading the code. We use the time the Suggestion Panel was in focus as a proxy for examination time; Figure 4.3 shows this time as a percentage of the total task duration. The No-LP group spent more time in the Suggestion Panel compared to LP for both Fixed-Prompt tasks. The difference is significant in the Pandas task ($p = .02, U = 11, median_{LP} = 14.05\%, median_{No-LP} = 30.47\%$) but not in Bigram ($p = .14, U = 20, median_{LP} = 24.70\%, median_{No-LP} = 36.57\%$). We also collected this data for the Open-Prompt tasks, even if this data should be interpreted with caution due to the unstructured nature of the tasks (*e.g.*, different participants invoked the assistant different numbers of times and got suggestions of different quality). The results are consistent with the Fixed-Prompt tasks—*i.e.*, No-LP participants spent more time in the Suggestion Panel—but the difference is not significant, and the effect in Box Plot is very small ($p = .14, U = 3.5, median_{LP} = 6.25\%, median_{No-LP} = 15.49\%$ for String Rewriting; $p = .67, U = 6, median_{LP} = 8.10\%, median_{No-LP} = 8.70\%$ for Box Plot).

Participants relied on runtime values more in API-heavy, one-off tasks. According to Figure 4.3, both groups spent more time examining the code in Bigram, while in Pandas they jumped to execution more immediately ($median_{\text{Pandas}} = 16.96\%, median_{\text{Bigram}} = 31.67\%, p = .04, U = 206$). This difference in validation strategies between the two tasks was also reflected in the interviews. For example, P1 described their strategy for Pandas as follows: *"I didn't look too closely in the actual code, I was just looking at the runtime values on the side."* Instead, in Bigram, participants cared more about the code itself, preferring suggestions based on their expected algorithm, data structure, or style (*e.g.* P15

"was really looking for the dictionary aspect"), with the most popular attribute being "short"/"readable", cited by 10 out of 17 participants. One explanation participants gave for the difference in behavior is that Pandas is an API-heavy task, and when dealing with unfamiliar APIs, reading the code is just not very helpful: *"When it's using more jargony stuff that doesn't translate directly into words in your brain, then seeing the preview makes it clearer"* (P3). Another explanation they gave is that Pandas was perceived by the participants as a *one-off* task, *i.e.*, it only needed to work on the one specified input, whereas Bigram was perceived as *general*, *i.e.*, it needed to work on *"any sort of string [...] not only [...] the specific string that was tested"* (P3); this was not explicit in the instructions, but in retrospect it is a reasonable assumption, given the problem domains and structure of the starter code. On the other hand, some LP participants conjectured that with more familiarity with Live Programming, they would rely on runtime values more, even in tasks like Bigram: *"If I were to use this tool again I would preview more immediately, just because I think I was very focused on whether it produced how I would solve the problem vs. whether it solved the problem correctly"* (P4).

LP participants benefitted from visualizing intermediate values. We looked into the validation strategies used in Bigram to identify the tie-resolution issue in AI suggestions (excluding P17 because they wrote the code from scratch). In the input we provided, it was hard to identify the most common bigram at a glance, which made it difficult to validate suggestions just by looking at the final result. *Five out of eight* LP participants found the issue by inspecting *intermediate values* and noticing that multiple bigrams in the input have the same count (the other three relied on custom test cases and code examination). In the No-LP group, three out of eight participants failed to identify the issue and of the remaining five who succeeded, *only one* (P6) relied on intermediate values to do so. In addition, multiple LP participants (P1, P3, P4) mentioned the usefulness of intermediate values in the interview, especially for long suggestions. P1 said: *"Because it's a block of text as a suggestion, having projection boxes is more important [...] my thought was 'let me go line by line to see what is going on'."* In contrast, a No-LP participant (P9) remarked that they *"had to really look through the code and try to visualize it in [their] mind."*

LP participants used liveness features for validation and debugging. For validation, LP participants made use of full liveness, *i.e.*, the ability to see the immediate effects of their edits. *Five* participants in Pandas added auxiliary calculations to double-check the correctness of the final output, *e.g.*, the mean

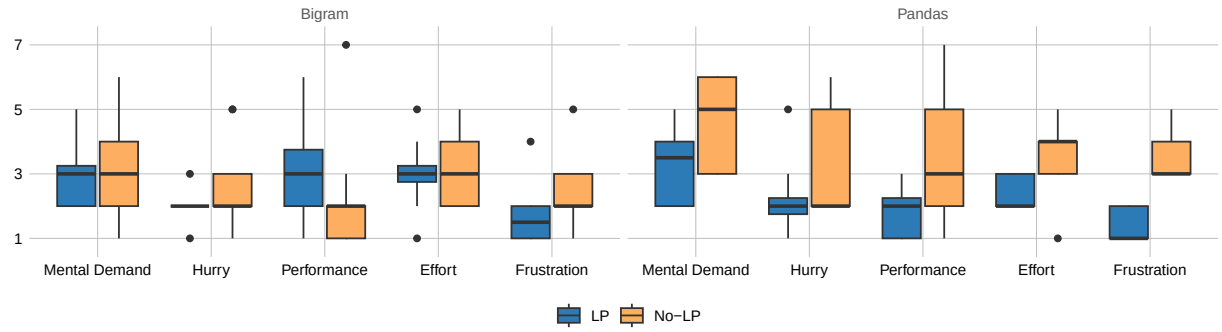


Figure 4.4: NASA Task Load Index (TLX) results for the Fixed-Prompt tasks: Bigram on the left, and Pandas on the right. Higher scores indicate higher cognitive load (in case of Performance this means higher failure rate).

of specific cells in the input table, comparing it to the output table. When it comes to debugging, LP participants made multiple rounds of trial and error guided by liveness. In fact, the example in Section 4.3 was inspired by P4's debugging process in the Bigram task. Also, in Box Plot, P1 made many repeated edits in an AI suggestion to tune the placement of a label, guided by error messages and incorrect outputs to figure out the precise usage of an unfamiliar API call. In the interview, they noted: *"I was definitely using the projections [...] as I was editing the suggestions to see if my intended changes actually were followed through."*

4.5.3 RQ3: Cognitive Load in Validation

LP participants experienced significantly lower cognitive load in the Pandas task but not the Bigram task.. In Pandas, we found that LP participants experienced significantly lower cognitive load in four out of five aspects of NASA-TLX [49]: mental demand ($p = .039, U = 14.5$), performance ($p = .048, U = 15.5$), effort ($p = .015, U = 11$), and frustration ($p = .0004, U = 0$). We find no significant differences in responses to the Bigram task, but LP participants reported slightly higher *performance* measures ($median_{LP} = 3, median_{No-LP} = 2$), which stand for higher failure rates.

No-LP participants found it hard to distinguish between the suggestions.. Participants from both the No-LP (P9, P14, P17) and LP (P3, P16) groups commented on the utility of seeing multiple suggestions at once: *"[Seeing multiple suggestions] gave me different ways to look at the code and gave me different ideas"* (P9) and *"multiple suggestions gave points of comparison that were useful"* (P14).

However, some No-LP participants (P6, P7, P15, P17) said they found the suggestions hard to distinguish. They noted the difficulty of differentiating just by reading the code because *"the suggestions [were] all almost the same thing"* (P7), and suggested that *"the tool did not really help with choosing between suggestions"* (P15). In comparison, some in the LP group (P1, P16) commented that Projection Boxes made selection easier; P1 said: *"Being able to preview, edit, and look at the projection boxes before accepting a snippet was very helpful when choosing between multiple suggestions."*

4.6 Discussion

Live Programming lowers the cost of validation by execution Although both LP and No-LP participants had access to runtime values as a validation mechanism, those without LP needed to examine the code to decide which values to print, add the `print` statements, run the code, and match each line in the output to the corresponding line in the code. If they wanted to inspect a different suggestion, they had to repeat this process from the start. Meanwhile, LP participants could simply click on the suggestion to preview it and get immediate access to all the relevant runtime information, easily switching between suggestions as necessary. In other words, LP lowers *the cost*—in terms of both time and mental effort—of access to runtime values. As a result, we saw LP participants relied on runtime values more for validation, as they spent less time examining the code in general—and significantly so for the Pandas task—and more often used intermediate values to find bugs in Bigram (**??**). Our findings are consistent with prior work [8, 69], which demonstrated that programmers more often use validation strategies with lower time costs. Hence, *by lowering the cost of access to runtime values, Live Programming promotes validation by execution.*

The lower cost of validation by execution prevents over- and under-reliance As discussed in Section 4.5.1, we found six instances of unsuccessful validation in our study, *all from the No-LP group*, over-relying on AI suggestions in the Bigram task, and under-relying in Pandas. We attribute these failures to the high cost of validation by execution: those who over-relied on the suggestions did not inspect the runtime behavior of the suggestions in enough detail, while those with under-reliance lacked

the affordances to do so effectively, and so ran out of time before they could validate the suggestions. Our results echo prior findings [119] that relate the cost of a validation strategy to its effectiveness in reducing over-reliance on AI. We conclude that *the lower cost of validation by execution in Live Programming leads to more accurate judgments of the correctness of AI-generated code.*

Validation strategies depend on the task Section 4.5.2 shows that participants overall spent significantly more time examining the code in Bigram than in Pandas and also paid more attention to code attributes in the former. Participants explained the difference in their validation strategies by two factors: (1) Pandas contained unfamiliar API calls, the meaning of which they could not infer from the code alone; and (2) they perceived Pandas as a one-off task, which only had to work on the given input. We conjecture that (1) is partly due to our participants being LP novices: as they get more used to the environment, they are likely to rely on previews more, even if they are not forced into it by an unfamiliar API (as P4 mentioned in Section 4.5.2). (2), though, is more fundamental: when dealing with a general task, correctness is not all that matters; code quality becomes important as well, and LP does not help with that.

In Open-Prompt tasks, code examination was less prevalent in the overall task duration, because in these tasks participants spent a significant amount of time on activities besides validation (*e.g.*, decomposing the problem and crafting prompts). It might seem surprising, however, that we did not see any difference in examination time between the two groups in Box Plot, which is an API-heavy, one-off task, similar to Pandas. This might be because, in Box Plot, the cost of validation by execution was already low for No-LP participants: this task did not require inspecting intermediate values, because the effects of each line of code were reflected on the final plot in a compositional manner (*i.e.*, it was easy to tell what each line of code was doing just by looking at the final plot).

In conclusion, *Live Programming does not completely eliminate the need for code examination but reduces it in tasks amenable to validation by execution.*

Live Programming lowers the cognitive load of validation by execution In Pandas, LP participants experienced lower cognitive load in four out of five TLX categories (Section 4.5.3). This confirms our hypotheses that LP lowers the cost of validation by execution, and that Pandas is a task amenable to

such validation. More specifically, we conjecture that, by automating away the process of writing `print` statements, LP reduces workflow interruptions, which were identified as one of the sources of increased cognitive load in reviewing AI-generated code [124].

In Bigram, however, we did not observe a similar reduction in cognitive load; in fact, LP participants reported *higher* cognitive load in the "performance" category (*i.e.*, they perceived themselves as less successful). Our interpretation is that the cognitive load in this task was dominated by debugging and not validation, and whereas all participants in the LP group engaged in debugging, only two-thirds of the No-LP group did so. Finally, the higher "performance" ratings from the LP group were from those who ran out of time trying to fix the code, and hence were aware that they had failed. These findings show that Live Programming by itself does not necessarily help with debugging a faulty suggestion. As we saw in Section 4.5.2, it can be helpful when the user has a set of potential fixes in mind, which they can quickly try out and get immediate feedback on. But when the user does not have potential fixes in mind, they need to rely on other tools, such as searching the web or using chat-based AI assistants.

From these findings, we conclude that *Live Programming lowers the cognitive load of validating AI suggestions when the task is amenable to validation by execution.*

4.7 Conclusion and Future Work

We investigated an application of Live Programming in the domain of AI-assisted programming, finding that LP can reduce over- and under-reliance on AI-generated code by lowering the cost of validation by execution. Our study is necessarily limited in scope: we focused on self-contained tasks due to LP's limited support for complex programs [114, 67] and its need for small demonstrative inputs [107]. We hope that our findings inform future studies on code validation and motivate further research into AI-LP integration. To that end, we highlight key opportunities below.

To offer liveness, LP places several burdens on the user. The user must provide a complete executable program and a set of test cases, and then look through potentially large runtime traces for the relevant information. AI may alleviate these burdens by filling in missing runtime values [108] for incomplete programs, generating test cases [64, 127], and predicting the most relevant information to be

displayed at each program point. Looking beyond the validation of newly generated code, there are also opportunities for AI-LP integration for debugging and code repair [127]. In combination, AI-LP would tighten the feedback loop of querying and repairing AI-generated code: users could validate code via LP, request repair using the runtime information from LP [36], and further validate the repair in LP.

Chapter 4, in part, is a reprint of the material as it will appear in the Proceedings of the 2024 CHI Conference on Human Factors and Computer Systems. Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, Sorin Lerner. ACM 2024. The author was a principal author and investigator on this work.

Conclusion

This dissertation has identified how program synthesizers can facilitate exploring for code, and that techniques to validate explore code are needed to reduce cognitive effort. This dissertation is only the first step on this path of research, with many possible paths in front of it. This conclusion highlights two of these paths: mode detection and LLM-powered exploration.

Mode Detection

As identified in Chapter 3, programmers using tools like Github Copilot [38] tend to oscillate between two different modes. At times they are being *accelerated* through their task, as their tool helps them put thought to code quickly. At other times, a programmer needs to *explore* a variety of options about their task, particularly when they are facing uncertainty. The way a programmer uses their code suggestion tool varies depending on what mode they are in.

Some exploratory tools already exist to help a user investigate implementation options [40, 41, 51, 130, 135]. However, these existing exploration tools are clunky and separate from the IDE. Programmers often quickly and fluidly switch between these two modes, making context switches highly distracting. While acceleration happens within an IDE, exploration tools are outside of the IDE. Developers are less likely to use a tool if it is not part of their IDE [72]. Whats more, a developer would need to *actively recognize* that they are exploring and then *actively choose* to use a separate tool.

Programmers would benefit if their tool could identify when they are likely trying to explore for code, and offer the opportunity to explore their design space. A number of different signals on their own, or together, could offer a predictor for when a programmer is exhibiting any number of non-acceleration behaviors. These other behaviors could be *validating* or *exploring*.

Typing Speed. During the study in Chapter 3, we observed that programmers tended to type in bursts. Typing quickly was often associated with being in acceleration mode. Meanwhile, those who were typing slowly tended to be talking or thinking through their code process. So, there may be multiple speeds at which a programmer types, and those different rates could be associated with different modes.

Repeated Edits. Much like how a writer will write and rewrite the same line to find the perfect phrasing, a programmer will occasionally rewrite the same line or block to optimize on readability, run time, or other factors. Repeated edits of the same line signal an opportunity to provide a range of suggestions. That edit history can also inform suggestions, to show either overlap or to use deleted lines as negative signals on suggestion content.

Pause Time. It is a common experience, to pause before writing a function call, as one tries to recall the function name and its arguments. When a programmer knows the name and arguments, there is little hesitation in writing down the function name and its arguments. So, a pause when a function or method call could be expected (*e.g.* after a dot access (.) in python), could be a signal of uncertainty for a function call. This uncertainty is an intervention opportunity to provide one or more suggestions.

Suggestion Accepts or Rejects. Programmers reject suggestions from Copilot or any code generating model for a number of reasons. Sometimes they are writing so quickly that a suggestion only appears for a fraction of a second. Other times, a programmer will specifically reject a suggestion with the escape button. signaling that the particular suggestion did not match their intention. This rejection does not *necessarily* mean no suggestion could match their intent, and it could be an opportunity, paired with a long pause to offer multiple alternatives.

Attention. When faced with an unknown in their code, some programmers turn their face away from the cursor, looking elsewhere for a solution. Attention, either through eye focus location, or general focus could be a useful signal. This signal could determine whether a programmer is in the middle of a thought process or looking for help. Such a signal may be useful in determining what interaction mode a user needs.

Any of these signals could be useful for determining an unobtrusive intervention opportunity. Done correctly, a tool offering the right kind of advice at the right time would feel similar to how programmers feel that Copilot, “completes my thoughts”, as some have said.

LLM Powered Code Exploration

Programmers do not always know exactly what they want to program, or how to go about writing it. When already using an AI assistant to program, coders will use it to overcome uncertainty, as seen in Chapter 3.

Currently, these assistants can provide up to 10 unique suggestions when asked. In their current form of a list, these suggestions provide little useful information and cannot often help answer a programmer's questions. These questions range from API usage to algorithmic to those of convention. A simple list of similar looking programs does little to answer these kinds of questions and consumes significant mental effort to eek that information out.

Suppose Ellen, a hobbyist programmer, is setting up a server at home to host her personal domain `home.ellen.com`. She needs code that automatically updates the DNS record for her domain to point to her dynamically assigned IP from her internet provider. She needs to use the Amazon AWS Route 53[4] service, with which she is not particularly familiar. Ellen uses an AI assistant to write part of her code for her, accepting the first suggestion she gets. Her code does not work the first time and now Ellen must debug code her AI assistant wrote for her.

Ellen's code does not contain a critical domain name adjustment. She needed to update `home.ellen.com.` (note the terminal dot). By chance the first suggestions did not provide this necessary terminal dot; however, if she looked through several suggestions she would have seen many suggestions add a `'.'` to her domain name. Without a way to get an overview of suggestions, Ellen could not see this small but necessary addition.

With a technique to provide an overview of suggestions, Ellen could learn common patterns, such as this dot addition, as well as common API calls, along with alternatives. But what information would be appropriate to surface to a user?

API names. It can be difficult to guess the correct API function call for a given task. Simply seeing two suggestions (*e.g.* `save()` or `save_record()`), in text provides little signal for which may be correct. A functional solution would likely appear in samples more frequently, so API name popularity can provide a useful information scent. Notably, there may not be enough text to parse or typecheck the

[4]https://aws.amazon.com/route53/

entire codeblock.

API arguments. The arguments and their required in a function are not always obvious, too. For example, a programmer may know that to use the AWS APIs, they need to provide an access key and a secret key, but the order of these is not always obvious. Showing common argument names across a distribution will show that one more commonly occurs before the other–in effect demonstrating a convention to the programmer.

Common subexpressions. Common subexpressions can highlight core nuggets of wisdom in the model. Properly setting the IP of the domain name requires a `'.'` suffix. A novice may not realize that in omitting this terminal dot, the record will be updated, but it will not work correctly. By seeing this subexpression multiple times, or seeing the frequency of this addition, a programmer could quickly learn something new about the API's usage, helping them write better code in the future.

Grouping of approaches. There is often not just one way to solve a programming task. That may be stringing together different sets of API calls to achieve the same end goal; or, using a recursive algorithm on a competitive programming task rather than an iterative one. There are tradeoffs with any implementation choice. To help a programmer explore the space of opportunities better, they need to first be *aware* that they eve have choice to begin with. Highlighting those top-level different approaches provides a set of starting places for those who may be stuck at the start of a task.

Highlight out-of-distribution information. In the same vein as highlighting those different approaches, is highlighting those less commonly selected approaches. Those less frequently sampled approaches may contain some valuable insight. For example, in completing this IP update task, a programmer may use any number of third party services to find their external IP. A majority of solutions call for using a particular `api.ipify.com` website. Only a couple suggestions of 100 use the Amazon ordained service for this task, `ipcheck.amazonaws.com`. A programmer way wish to reduce the number of external dependencies, and not use the most common, or second-most common service. Instead, that single suggestion with an Amazon provided url may provide more utility.

In conclusion, these two future directions of research, mode detection and LLM powered code exploration, will aid programmers in more aspects of their workflow, powered by probabilistic suggestion engines. Mode detection will improve how a tool works with its user, helping them explore when they

need it. LLM powered code exploration will deepen those techniques, and will need validation alongside it. This thesis has that both formal and probabilistic program synthesizers can aid in this exploratory process of programming, and that techniques to validate that explored code reduce the cognitive effort.

Acknowledgements. The conclusion, in part, is a reprint of the material, current being prepared for submission for publication, with the following coauthors: Michael B. James, Emmanuel Anaya Gonzalez, Mark Barbone, Elena L. Glassman, Nadia Polikarpova. The author was a principal investigator on this work.

Bibliography

[1] ALBARGHOUTHI, A., GULWANI, S., AND KINCAID, Z. *Recursive Program Synthesis*, vol. 8044. Springer Berlin Heidelberg, 2013, p. 934950.

[2] ALUR, R., RADHAKRISHNA, A., AND UDUPA, A. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (2017), pp. 319–336.

[3] AMAZON. Codewhisperer. `https://aws.amazon.com/codewhisperer/`, 2023.

[4] AN, J. D., CHAUDHURI, A., FOSTER, J. S., AND HICKS, M. Dynamic inference of static types for ruby. In *POPL. Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 459–472.

[5] AUGUSSTSON, L. Djinn. `https://github.com/augustss/djinn`, 2005.

[6] BALOG, M., GAUNT, A. L., BROCKSCHMIDT, M., NOWOZIN, S., AND TARLOW, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[7] BARKE, S., JAMES, M. B., AND POLIKARPOVA, N. Grounded Copilot: How Programmers Interact with Code-Generating Models, Mar. 2023.

[8] BARKE, S., JAMES, M. B., AND POLIKARPOVA, N. Grounded Copilot: How programmers interact with code-generating models. *PACMPL 7*, OOPSLA (2023).

[9] BIRD, C., FORD, D., ZIMMERMANN, T., FORSGREN, N., KALLIAMVAKOU, E., LOWDERMILK, T., AND GAZIT, I. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue 20*, 6 (Dec. 2022), 35–57.

[10] BONNAIRE-SERGEANT, A. *Typed Clojure in Theory and Practice*. PhD thesis, Indiana University, Bloomington, 2019.

[11] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models

are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[12] CABRERA, L., MALONEY, J. H., AND WEINTROP, D. Programs in the Palm of your Hand: How Live Programming Shapes Children's Interactions with Physical Computing Devices. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children* (Boise ID USA, June 2019), ACM, pp. 227–236.

[13] CAMPUSANO, M., BERGEL, A., AND FABRY, J. Does Live Programming Help Program Comprehension? – A user study with Live Robot Programming. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (Amsterdam Netherlands, Nov. 2016), ACM, p. 8.

[14] CARLSTON, D. E. *Dual-Process Theories*. 2013.

[15] CARVER, J. The impact of background and experience on software inspections. *Empirical Softw. Engg. 9*, 3 (sep 2004), 259–262.

[16] CHARMAZ, K. *Constructing grounded theory*. sage, 2014.

[17] CHASINS, S. E., MUELLER, M., AND BODIK, R. Rousillon: Scraping distributed hierarchical web data. In *UIST 2018* (2018), pp. 963–975.

[18] CHASINS, S. E., MUELLER, M., AND BODIK, R. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2018), UIST '18, Association for Computing Machinery, pp. 963–975.

[19] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., DE OLIVEIRA PINTO, H. P., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., RAY, A., PURI, R., KRUEGER, G., PETROV, M., KHLAAF, H., SASTRY, G., MISHKIN, P., CHAN, B., GRAY, S., RYDER, N., PAVLOV, M., POWER, A., KAISER, L., BAVARIAN, M., WINTER, C., TILLET, P., SUCH, F. P., CUMMINGS, D., PLAPPERT, M., CHANTZIS, F., BARNES, E., HERBERT-VOSS, A., GUSS, W. H., NICHOL, A., PAINO, A., TEZAK, N., TANG, J., BABUSCHKIN, I., BALAJI, S., JAIN, S., SAUNDERS, W., HESSE, C., CARR, A. N., LEIKE, J., ACHIAM, J., MISRA, V., MORIKAWA, E., RADFORD, A., KNIGHT, M., BRUNDAGE, M., MURATI, M., MAYER, K., WELINDER, P., MCGREW, B., AMODEI, D., MCCANDLISH, S., SUTSKEVER, I., AND ZAREMBA, W. Evaluating large language models trained on code, 2021.

[20] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., PINTO, H. P. D. O., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., ET AL. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[21] CHUGH, R., HEMPEL, B., SPRADLIN, M., AND ALBERS, J. Programmatic and direct manipulation, together at last. In *PLDI '16* (New York, NY, USA, 2016), PLDI '16, Association for Computing Machinery, p. 341354.

[22] CHUGH, R., LERNER, S., AND JHALA, R. Type inference with run-time logs. In *Workshop on Scripts to Programs (STOP)* (2011).

[23] CLAESSEN, K., AND HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00* (Sep 2000), ICFP '00, Association for Computing Machinery, p. 268279.

[24] CLAESSEN, K., AND HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (Sep 2000), ICFP'00, Association for Computing Machinery, pp. 268–279.

[25] DANIELSSON, N. A., AND JANSSON, P. Chasing bottoms. In *Mathematics of Program Construction* (2004), D. Kozen, Ed., Lecture Notes in Computer Science, Springer, p. 85109.

[26] DELINE, R., AND FISHER, D. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (Atlanta, GA, Oct. 2015), IEEE, pp. 111–119.

[27] DELINE, R. A. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama Japan, May 2021), ACM, pp. 1–11.

[28] DI COSMO, R. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming 3*, 3 (1993), 485–525. Special Issue on ML.

[29] DROSOS, I., BARIK, T., GUO, P. J., DELINE, R., AND GULWANI, S. Wrex: A unifed programming-by-example interaction for synthesizing readable code for data scientists. 12.

[30] DROSOS, I., BARIK, T., GUO, P. J., DELINE, R., AND GULWANI, S. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *CHI 2020* (New York, NY, USA, 2020), Association for Computing Machinery, p. 112.

[31] DROSOS, I., BARIK, T., GUO, P. J., DELINE, R., AND GULWANI, S. *Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists*. Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–12.

[32] EFIRD, J. Blocked randomization with randomly selected block sizes. *International journal of environmental research and public health 8*, 1 (2011), 15–20.

[33] FENG, Y., MARTINS, R., WANG, Y., DILLIG, I., AND REPS, T. W. Component-based synthesis for complex apis. In *POPL 2017* (2017).

[34] FERDOWSIFARD, K., BARKE, S., PELEG, H., LERNER, S., AND POLIKARPOVA, N. Loopy: Interactive program synthesis with control structures. *Proc. ACM Program. Lang. 5*, OOPSLA (oct 2021).

[35] FERDOWSIFARD, K., ORDOOKHANIANS, A., PELEG, H., LERNER, S., AND POLIKARPOVA, N. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2020), Association for Computing Machinery, pp. 614–626.

[36] FERDOWSIFARD, K., ORDOOKHANIANS, A., PELEG, H., LERNER, S., AND POLIKARPOVA, N. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2020), UIST '20, Association for Computing Machinery, p. 614626.

[37] FRANKLE, J., OSERA, P.-M., WALKER, D., AND ZDANCEWIC, S. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016), POPL 16, ACM, p. 802815. event-place: St. Petersburg, FL, USA.

[38] FRIEDMAN, N., Jun 2021.

[39] GLASER, B. G., AND STRAUSS, A. L. *The discovery of grounded theory: strategies for qualitative research*, 5. paperback print ed. Aldine Transaction, 1967.

[40] GLASSMAN, E. L., SCOTT, J., SINGH, R., GUO, P. J., AND MILLER, R. C. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction 22*, 2 (Mar 2015), 7:1–7:35.

[41] GLASSMAN, E. L., ZHANG, T., HARTMANN, B., AND KIM, M. Visualizing api usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Apr 2018), ACM, p. 112.

[42] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), pp. 317–330.

[43] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan 2011), POPL 11, Association for Computing Machinery, p. 317330.

[44] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Component-based synthesis applied to bitvector programs. 14.

[45] GUO, D., SVYATKOVSKIY, A., YIN, J., DUAN, N., BROCKSCHMIDT, M., AND ALLAMANIS, M. Learning to complete code with sketches. In *International Conference on Learning Representations* (2021).

[46] GUO, Z., JAMES, M., JUSTO, D., ZHOU, J., WANG, Z., JHALA, R., AND POLIKARPOVA, N. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang. 4*, POPL (2020), 12:1–12:28.

[47] GVERO, T., KUNCAK, V., KURAJ, I., AND PISKAC, R. Complete completion using types and weights. In *PLDI 2013* (2013).

[48] HANCOCK, C. M. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[49] HART, S. G., AND STAVELAND, L. E. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, vol. 52. Elsevier, 1988, pp. 139–183.

[50] HEINEMAN, G. T., BESSAI, J., DÜDDER, B., AND REHOF, J. A long and winding road towards modular synthesis. In *ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I* (2016), pp. 303–317.

[51] HOLLOWAY, T., SWOOPES, C., ARAWJO, I., PELEG, H., AND GLASSMAN, E. Reverse sketching.

[52] HUANG, R. L., FERDOWSI, K., SELVARAJ, A., SOOSAI RAJ, A. G., AND LERNER, S. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, Feb. 2022), SIGCSE 2022, Association for Computing Machinery, pp. 495–501.

[53] HUGHES, J., AND ORCHARD, D. Resourceful program synthesis from graded linear types. 54.

[54] JAMES, M. B., GUO, Z., WANG, Z., DOSHI, S., PELEG, H., JHALA, R., AND POLIKARPOVA, N. Digging for fold: Synthesis-aided api discovery for haskell. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[55] JAYAGOPAL, D., LUBIN, J., AND CHASINS, S. E. Exploring the learnability of program synthesizers by novice programmers. 15.

[56] JHA, S., GULWANI, S., SESHIA, S. A., AND TIWARI, A. Oracle-guided component-based program synthesis. In *ICSE '10* (2010), vol. 1, ACM Press, p. 215.

[57] JIANG, E., TOH, E., MOLINA, A., OLSON, K., KAYACIK, C., DONSBACH, A., CAI, C. J., AND TERRY, M. Discovering the syntax and strategies of natural language programming with generative language models. In *CHI Conference on Human Factors in Computing Systems* (2022), pp. 1–19.

[58] KAHNEMAN, D. *Thinking, fast and slow*. Penguin psychology. Penguin Books, 2011.

[59] KALYAN, A., MOHTA, A., POLOZOV, O., BATRA, D., JAIN, P., AND GULWANI, S. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018).

[60] KANG, H., AND GUO, P. J. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City QC Canada, Oct. 2017), ACM, pp. 737–745.

[61] KITE. Kite: Ai-powered completions for jupyterlab. `https://www.kite.com/integrations/jupyter/`, 2020.

[62] KNOTH, T., WANG, D., POLIKARPOVA, N., AND HOFFMANN, J. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Jun 2019), PLDI 2019, Association for Computing Machinery, p. 253268.

[63] KRAMER, J.-P., KURZ, J., KARRER, T., AND BORCHERS, J. How Live Coding Affects Developers Coding Behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (July 2014), pp. 5–8. ISSN: 1943-6106.

[64] LAHIRI, S. K., FAKHOURY, S., NAIK, A., SAKKAS, G., CHAKRABORTY, S., MUSUVATHI, M., CHOUDHURY, P., VON VEH, C., INALA, J. P., WANG, C., AND GAO, J. Interactive code generation via test-driven user-intent formalization, 2023.

[65] LE, V., PERELMAN, D., POLOZOV, O., RAZA, M., UDUPA, A., AND GULWANI, S. Interactive program synthesis. *CoRR abs/1703.03539* (2017).

[66] LERNER, S. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Apr 2020), ACM, pp. 1–7.

[67] LERNER, S. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2020), CHI '20, Association for Computing Machinery, p. 17.

[68] LI, Y., CHOI, D., CHUNG, J., KUSHMAN, N., SCHRITTWIESER, J., LEBLOND, R., ECCLES, T., KEELING, J., GIMENO, F., LAGO, A. D., HUBERT, T., CHOY, P., D'AUTUME, C. D. M., BABUSCHKIN, I., CHEN, X., HUANG, P.-S., WELBL, J., GOWAL, S., CHEREPANOV, A., MOLLOY, J., MANKOWITZ, D. J., ROBSON, E. S., KOHLI, P., DE FREITAS, N., KAVUKCUOGLU, K., AND VINYALS, O. Competition-level code generation with alphacode, 2022.

[69] LIANG, J. T., YANG, C., AND MYERS, B. A. Understanding the usability of ai programming assistants, 2023.

[70] LUBIN, J., AND CHASINS, S. E. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages 5*, OOPSLA (Oct 2021), 1–30.

[71] LUBIN, J., COLLINS, N., OMAR, C., AND CHUGH, R. Program sketching with live bidirectional evaluation. *arXiv:1911.00583 [cs]* (May 2020). arXiv: 1911.00583.

[72] MAALEJ, W., TIARKS, R., ROEHM, T., AND KOSCHKE, R. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology 23*, 4 (Sep 2014), 31:1–31:37.

[73] MANDELIN, D., XU, L., BODÍK, R., AND KIMELMAN, D. Jungloid mining: Helping to navigate the api jungle. In *PLDI 2005* (2005).

[74] MILLI, S., LIEDER, F., AND GRIFFITHS, T. L. A rational reinterpretation of dual-process theories. *Cognition 217* (2021), 104881.

[75] MILTNER, A., GULWANI, S., LE, V., LEUNG, A., RADHAKRISHNA, A., SOARES, G., TIWARI, A., AND UDUPA, A. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–29.

[76] MILTNER, A., MAINA, S., FISHER, K., PIERCE, B. C., WALKER, D., AND ZDANCEWIC, S. Synthesizing symmetric lenses. *Proc. ACM Program. Lang. 3*, ICFP 2019 (July 2019).

[77] MITCHELL, N. Hoogle. `https://www.haskell.org/hoogle/`, 2004.

[78] MOZANNAR, H., BANSAL, G., FOURNEY, A., AND HORVITZ, E. Reading between the lines: Modeling user behavior and costs in ai-assisted programming, 2022.

[79] NI, W., SUNSHINE, J., LE, V., GULWANI, S., AND BARIK, T. 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* (2021), pp. 258–269.

[80] NORELL, U. Dependently typed programming in agda. In *AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures* (2008), pp. 230–266.

[81] OPENAI. Chatgpt. `https://chat.openai.com/`, 2023.

[82] OPENAI. Gpt-3.5. `https://platform.openai.com/docs/models/gpt-3-5`, 2023.

[83] OSERA, P., AND ZDANCEWIC, S. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), pp. 619–630.

[84] OSERA, P.-M., AND ZDANCEWIC, S. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), PLDI 15, ACM, p. 619630. event-place: Portland, OR, USA.

[85] PELEG, H., GABAY, R., ITZHAKY, S., AND YAHAV, E. Programming with a read-eval-synth loop. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[86] PELEG, H., AND POLIKARPOVA, N. Perfect is the enemy of good: Best-effort program synthesis. 30.

[87] PELEG, H., SHOHAM, S., AND YAHAV, E. Programming not only by example. In *ICSE 2018* (2018), ACM, pp. 1114–1124.

[88] PELEG, H., SHOHAM, S., AND YAHAV, E. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering* (2018), ICSE '18, ACM, pp. 1114–1124. tex.ids: pelegProgrammingNotOnly2018a event-place: Gothenburg, Sweden.

[89] PENNINGTON, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology 19*, 3 (Jul 1987), 295–341.

[90] PERELMAN, D., GULWANI, S., BALL, T., AND GROSSMAN, D. Type-directed completion of partial expressions. In *PLDI '12, Beijing, China - June 11 - 16, 2012* (2012), pp. 275–286.

[91] PERRY, N., SRIVASTAVA, M., KUMAR, D., AND BONEH, D. Do users write more insecure code with ai assistants?, 2022.

[92] PHOTHILIMTHANA, P. M., THAKUR, A., BODIK, R., AND DHURJATI, D. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar 2016), ASPLOS 16, Association for Computing Machinery, p. 297310.

[93] PLOTKIN, G. *Lattice Theoretic Properties of Subsumption*. Edinburgh University, Department of Machine Intelligence and Perception, 1970.

[94] POLIKARPOVA, N., KURAJ, I., AND SOLAR-LEZAMA, A. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (2016), pp. 522–538.

[95] POLIKARPOVA, N., AND SERGEY, I. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan 2019), 72:1–72:30.

[96] POLOZOV, O., AND GULWANI, S. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices 50*, 10 (2015), 107–126.

[97] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language models. *SIGPLAN Not. 49*, 6 (June 2014), 419–428.

[98] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 419–428.

[99] REYNOLDS, J. C. Transformational systems and the algebraic structure of atomic formulas.

[100] ROSS, S. I., MARTINEZ, F., HOUDE, S., MULLER, M., AND WEISZ, J. D. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development, Feb. 2023. arXiv:2302.07080 [cs].

[101] RUNCIMAN, C., NAYLOR, M., AND LINDBLAD, F. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell Symposium 2008* (Sep 2008), Haskell '08, Association for Computing Machinery, p. 3748.

[102] SALVANESCHI, G., PROKSCH, S., AMANN, S., NADI, S., AND MEZINI, M. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering 43*, 12 (Dec 2017), 11251143.

[103] SAMSI, S., ZHAO, D., MCDONALD, J., LI, B., MICHALEAS, A., JONES, M., BERGERON, W., KEPNER, J., TIWARI, D., AND GADEPALLY, V. From words to watts: Benchmarking the energy costs of large language model inference. arXiv:2310.03003 [cs].

[104] SARKAR, A., GORDON, A. D., NEGREANU, C., POELITZ, C., RAGAVAN, S. S., AND ZORN, B. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[105] SASNAUSKAS, R., CHEN, Y., COLLINGBOURNE, P., KETEMA, J., LUP, G., TANEJA, J., AND REGEHR, J. Souper: A synthesizing superoptimizer. *arXiv:1711.04422 [cs]* (Apr 2018). arXiv: 1711.04422.

[106] SINGH, R., AND GULWANI, S. Predicting a correct program in programming by example. In *CAV - 27th International Conference, 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I* (2015), pp. 398–414.

[107] SOARES, G., MURPHY-HILL, E., AND GHEYI, R. Live feedback on behavioral changes. In *2013 1st International Workshop on Live Programming (LIVE)* (2013), pp. 23–26.

[108] SOUZA, B., AND PRADEL, M. LExecutor: Learning-guided execution, 2023.

[109] SRIVASTAVA, S., GULWANI, S., AND FOSTER, J. S. From program verication to program synthesis. 14. tex.ids: srivastavaProgramVerificationProgram, srivastavaProgramVerificationPrograma.

[110] STOL, K.-J., RALPH, P., AND FITZGERALD, B. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering* (May 2016), ICSE '16, Association for Computing Machinery, pp. 120–131.

[111] STRAUSS, A. L., AND CORBIN, J. *Basics of Qualitative Reseach: Grounded Theory Procedures and Techniques*. SAGE Publications, Inc., 1990.

[112] TABNINE. Code faster with AI completions. https://www.tabnine.com/, 2023.

[113] TANG, N., CHEN, M., NING, Z., BANSAL, A., HUANG, Y., MCMILLAN, C., AND LI, T. J.-J. An empirical study of developer behaviors for validating and repairing ai-generated code. Plateau Workshop.

[114] TANIMOTO, S. L. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)* (2013), IEEE, pp. 31–34.

[115] TAO, Y., DANG, Y., XIE, T., ZHANG, D., AND KIM, S. 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* (Nov 2012), FSE 12, Association for Computing Machinery, p. 111.

[116] URZYCZYN, P. Inhabitation in typed lambda-calculi (A syntactic approach). In *TLCA '97, Nancy, France, April 2-4, 1997, Proceedings* (1997), pp. 373–389.

[117] VAITHILINGAM, P., ZHANG, T., AND GLASSMAN, E. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Late-Breaking Work* (2022).

[118] VASCONCELOS, H., BANSAL, G., FOURNEY, A., LIAO, Q. V., AND VAUGHAN, J. W. Generation probabilities are not enough: Improving error highlighting for ai code suggestions. In *HCAI Workshop at NeurIPS* (2022).

[119] VASCONCELOS, H., JÖRKE, M., GRUNDE-MCLAUGHLIN, M., GERSTENBERG, T., BERNSTEIN, M., AND KRISHNA, R. Explanations Can Reduce Overreliance on AI Systems During Decision-Making, Jan. 2023. arXiv:2212.06823 [cs].

[120] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. arXiv:1706.03762 [cs].

[121] VICTOR, B. Learnable Programming, 2012.

[122] WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad-hoc. In *POPL 1989, Austin, Texas, USA, January 11-13, 1989* (1989), pp. 60–76.

[123] WANG, C., FENG, Y., BODIK, R., DILLIG, I., CHEUNG, A., AND KO, A. J. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), CHI '21, Association for Computing Machinery.

[124] WANG, R., CHENG, R., FORD, D., AND ZIMMERMANN, T. Investigating and designing for trust in ai-powered code generation tools. *arXiv preprint arXiv:2305.11248* (2023).

[125] WASTL, E. Advent of code. https://adventofcode.com/2021, 2021.

[126] WEISZ, J. D., MULLER, M., HOUDE, S., RICHARDS, J., ROSS, S. I., MARTINEZ, F., AGARWAL, M., AND TALAMADUPULA, K. Perfection not required? human-ai partnerships in code translation. In *26th International Conference on Intelligent User Interfaces* (New York, NY, USA, 2021), Association for Computing Machinery, pp. 402–412.

[127] WILSON-THOMAS, M. Simplified code refinement and debugging with github copilot chat, Aug. 2023.

[128] XU, F. F., JIANG, Z., YIN, P., VASILESCU, B., AND NEUBIG, G. Incorporating external knowledge through pre-training for natural language to code generation, 2020.

[129] XU, F. F., VASILESCU, B., AND NEUBIG, G. In-ide code generation from natural language: Promise and challenges, 2021.

[130] YAN, L., KIM, M., HARTMANN, B., ZHANG, T., AND GLASSMAN, E. L. Concept-annotated examples for library comparison. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Oct 2022), ACM, p. 116.

[131] YANOW, D. Qualitative-interpretive methods in policy research. In *Handbook of public policy analysis*. Routledge, 2017, pp. 431–442.

[132] ZHANG, T., CHEN, Z., ZHU, Y., VAITHILINGAM, P., WANG, X., AND GLASSMAN, E. L. Interpretable program synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), CHI '21, Association for Computing Machinery.

[133] ZHANG, T., LOWMANSTONE, L., WANG, X., AND GLASSMAN, E. L. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Oct 2020), ACM, p. 627648.

[134] ZHANG, T., LOWMANSTONE, L., WANG, X., AND GLASSMAN, E. L. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2020), Association for Computing Machinery, pp. 627–648.

[135] ZHANG, T., YANG, D., LOPES, C., AND KIM, M. Analyzing and supporting adaptation of online code examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (May 2019), IEEE, p. 316327.

[136] ZHAO, C., SHEN, I.-C., FUKUSATO, T., KATO, J., AND IGARASHI, T. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces* (New York, NY, USA, Mar. 2022), IUI '22, Association for Computing Machinery, pp. 392–404.

[137] ZHOU, X., BODIK, R., CHEUNG, A., AND WANG, C. Synthesizing analytical sql queries from computation demonstration. In *PLDI* (2022).