UNIVERSITY OF CALIFORNIA SAN DIEGO

Human-Centered Program Synthesis

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

in

Computer Science

by

Kasra Ferdowsifard

2024

EPIGRAPH

”Oh, no,” said the Lecturer in Recent Runes, pushing his chair back.
”Not that. That’s meddling with things you don’t understand.”
”Well, we are wizards,” said Ridcully.
”We’re supposed to meddle with things we don’t understand.
If we hung around waitin’ till we understood things we’d never get anything done.”

Interesting Times – Terry Pratchett

Humans make everything needlessly difficult.

A Long Way to a Small Angry Planet – Becky Chambers

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professors Nadia Polikarpova, Sorin Lerner and Hila Peleg for their kindness, patience, and peerless mentorship and advice. An equal thanks goes to my family, whose support and trust in me was invaluable as I went through the ups and down of grad school. And, of course, a huge thanks to my friends Adam, Alex, Ange, Aidan, Asaiah, Chester, Colton, Janet, Marcus, Mark and Zeph. Grad school through the pandemic was an exceptionally depressing and isolating experience, and our many dinners, movie and game nights were essential in keeping my sanity (mostly) intact. Finally, I would like to thank UAW 2865, who gave me hope and power when I needed it most.

Chapter 2, in full, is a reprint of Small-Step Live Programming by Example, as it appears in the Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20). It was done in collaboration with Allen Ordookhanians, Hila Peleg, Sorin Lerner and Nadia Polikarpova. The dissertation author was a primary investigator and author of this paper.

Chapter 3, in full, is a reprint of LooPy: interactive program synthesis with control structures, as it appears in the Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA. It was done in collaboration with Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. The dissertation author was a primary investigator and author of this paper.

Chapter 4, in part, is currently being prepared for submission for publication of the material. It was done in collaboration with Hila Peleg. The dissertation author was a primary investigator and author of this material.

Chapter 5, in full, is a reprint of Validating AI-Generated Code with Live Programming, as it appears in the Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24). It was done in collaboration with Ruanqianqian (Lisa) Huang, Michael B. James, Sorin Lerner and Nadia Polikarpova. The dissertation author was a primary investigator and co-author of this material.

Chapter 6, in full, is a reprint of ColDeco An End User Spreadsheet Inspection Tool for AI-Generated Code, as it appears in the 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). It was done in collaboration with Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar and Benjamin Zorn. The dissertation author was a primary investigator and author of this paper.

VITA

2017	Associate of Science for Transfer, Mathematics Diablo Valley College
2017	Associate of Science for Transfer, Physics Diablo Valley College
2019	Bachelor of Science, Electrical Engineering and Computer Science University of California, Berkeley
2022	Master of Science, Computer Science University of California San Diego
2024	Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2024. *Validating AI Generated Code with Live Programming*. In Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3613904.3642495.

Kasra Ferdowsi, Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar, and Benjamin Zorn. 2023. *ColDeco: An End User Spreadsheet Inspection Tool for AI-Generated Code*. 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Washington, DC, USA, 82-91. https://doi.org/10.1109/VL-HCC57772.2023.00017.

Kasra Ferdowsi. 2023. *Towards Human-Centered Types & Type Debugging*. Plateau Workshop. https://doi.org/10.1184/R1/22227457.v1

Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. *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 - Volume 1 (SIGCSE 2022), Vol. 1. Association for Computing Machinery, New York, NY, USA, 495–501. https://doi.org/10.1145/3478431.3499305

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. *LooPy: Interactive Program Synthesis with Control Structures*. Proc. ACM Program. Lang. 5, OOPSLA, Article 153 (October 2021), 29 pages. https://doi.org/10.1145/3485530

Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. *Small-Step Live Programming by Example*. In Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20). Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

ABSTRACT OF THE DISSERTATION

Human-Centered Program Synthesis

by

Kasra Ferdowsifard

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Sorin Lerner, Co-Chair
Professor Nadia Polikarpova, Co-Chair

The intelligent programming assistant, directly integrated into the programmer's workflow, has been a long time dream of programmers and researchers alike. Before the advent of Large Language Models (LLMs), this dream came closest to reality with Program Synthesis in the form of Programming-by-Example, and now tools such as GitHub Copilot have brought a version of it using LLMs to consumers. Alongside the technical developments in Program Synthesis and LLMs that made these tools possible, another area of research has focused on the usability of such tools, investigating user interfaces and interaction models that could most effectively employ these techniques to the benefit of programmers.

This dissertation explores the overlap between these research areas, focusing on synchronous program synthesis where the user is in-the-loop, specifying synthesis problems and waiting on the result. It includes human-centered contributions to each step the synthesis process, exploring the use of Live Programming to improve specification in Programming-by-Example in SNIPPY and LOOPY, an algorithmic contribution to bottom-up enumerative synthesis with side effects in SOBEQ, and exploring interfaces for validation of AI-generated programs for experienced developers in LEAP, and end user programmers in COLDECO.

Chapter 1

Introduction

Synthesizing programs from some form of specification has existed since at least the 1960s [184], and is an active area of research today [103, 14, 70, 57, 82, 49, 69, 55, 53, 108]. At time of writing, at least two forms of program synthesis have been developed into commercial tools: FlashFill [63], which brought Programming-by-Example (PBE) to end-user programmers in spreadsheets, and AI-powered auto-complete tools such as GitHub Copilot [60] and TabNine [171], which use code context and natural language to generate many lines of code directly in the IDE. Alongside such tools and techniques, a rich literature has been developing which explores the human-centered design space of program synthesis, and addresses its many usability challenges [87, 76, 201, 179, 13, 110, 129, 190, 141, 143].

This dissertation explores a particular approach to designing user interactions for many such synthesis tools, focused on the use of Live Programming (LP). So, in this introduction, I will first provide the necessary background, followed by an overview of the chapters of this dissertation.

1.1 Background

In this section, I will first cover some existing works on human-centered program synthesis and introduce the necessary terminology for situating this dissertation within that body of work. Then I will briefly discuss live programming, including the LP environment used in this dissertation, and what makes LP distinct from other programming environments such as REPLs and computational notebooks.

1.1.1 Program Synthesis

Program synthesis as a whole is too broad a topic for the scope of this introduction[1]. But, from a human-centered perspective, synthesis tools can be generally split into two forms of intended interaction: *synchronous* or *asynchronous*. In *synchronous* synthesis, the user is in-the-loop, waits for the synthesis results, and can be expected to accept, reject or otherwise offer feedback to the synthesizer during the synthesis process. *Asynchronous* synthesis instead assumes the form of a background process, and while a human may be involved in providing its specification and validating its results, they are not assumed to be present while synthesis is in progress.

Most "interactive" program synthesis [202, 89, 57, 202] and AI-powered tools [60, 171, 96] are synchronous, having short timeouts and expecting the user to be present to interact with the system. Traditional synthesis tools in Programming Languages (PL) research instead either ignore the question of how a user is intended to interact with them, or apply to domains where synthesis does not require a user in the loop, starting with existing specifications and verifying solution correctness with a solver (*e.g.*, LENS [145]). I will touch on designs for asynchronous synthesis in Chapter 7, but this dissertation focuses on synchronous synthesis. So, to contextualize its contributions, I will next introduce the three stages in the synchronous program synthesis interaction model.

From the user's perspective, synchronous synthesis consists of three separate stages, which you can see in Fig. 1.1. First, the user must *specify* the synthesis problem or task, then the tool executes its particular synthesis *algorithm* to synthesize a program matching the given specification, and finally (assuming that it succeeds) the user must *validate* the solution.

Specification. For general purpose programming, specification usually takes the form of input-

[1]For a more thorough overview of program synthesis and a discussion of its technical dimensions see [67].

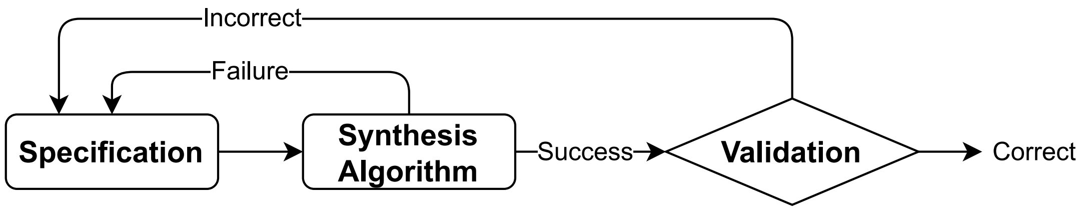


Figure 1.1. The stages of synchronous program synthesis. Respecifying and repairing incorrect solutions may be considered separate stages, but here they are folded into "Specification" and "Validation" respectively.

output examples (also called Programming-by-Example or PBE) [63] or natural language [30], but it can be as diverse as observed refactoring edits [126], desired syntactic features [143, 141], and function types [55]. And, of course, domain-specific synthesis can leverage specialized interfaces such as direct manipulation for SVGs [77], a visualization editor [186], and specifying operations on an input table [17].

The first three chapters of this dissertation focus on PBE, where the user provides the specification in the form of input-output examples. SNIPPY and LOOPY (Chapter 2 and Chapter 3) streamline providing examples by automatically including the input environment, and asking the user to only provide their desired outputs, while SOBEQ (Chapter 4) also includes affordances for specifying the desired side-effects (or lack thereof). The last chapters, which explore AI-generated programs, use natural language text and code context as the specification, using input examples for validation instead.

Synthesis Algorithm. While there is a wealth of works on developing synthesis algorithms, they are almost always opaque to the user. This is not necessarily a limitation, since the user of a synthesis tool should ideally be able to use it effectively without understanding the underlying algorithm. Though some have explored addressing algorithm transparency with interpretable program synthesis [201] for enumerative techniques[2].

Gulwani et al. [67] categorize synthesis algorithms into four groups, plus combinations

[2]As far as I am aware, while Explainable AI is a rich field of research, there have been no publications on explainable AI for LLM-powered code generation specifically.

of the four. These include *enumerative search*, where programs are enumerated in some order, and checked to see if they satisfy the specification, *deductive search* which recursively breaks down the specification top-down, *constraint solving* approaches which first generate a logical constraint from the specification and use a constraint solver to solve it, and finally *statistical* methods including using probabilistic grammars and machine learning.

Including the algorithm as a key stage in synthesis interaction may seem strange, since this is the step where user interaction is almost never directly available. However, as Peleg [140] has argued before, the synthesis algorithm and interaction model are tightly coupled, affecting the interaction design at every other stage of synthesis. Not only does the algorithm need to meet users' needs and expectations, but it determines many design considerations including the forms of meaningful specification that the user can or is required to provide, the total time the user needs to wait before getting the results (making certain algorithms unsuitable to synchronous synthesis entirely), if and how the tool can communicate reasons for failure to the user, the number of possible programs the user needs to inspect, and the kinds of guarantees those solutions provide to name a few.

This dissertation includes a human-centered exploration of both enumerative and AI-driven synthesis algorithms, though the only directly algorithmic contributions are to bottom-up enumerative synthesis with control structures (Chapter 3) and what is to my knowledge the first bottom-up proof-directed synthesis algorithm (Chapter 4).

Validation. Depending on the synthesis algorithm, a synthesizer may produce anywhere from a single proven-correct program to a set of incorrect solutions. And, in contexts such as general purpose programming, a solution may be unsuitable for reasons other than correctness (readability, maintainability, etc.) [110]. Thus the user may be required to validate[3] the synthesis result, either by checking that the one solution is suitable, or choosing between a set of programs.

[3] I will use the term "validation", but others have used "recognition" [86], "verifying" [129], and "co-audit" [61] to refer to similar concepts.

The forms of validation assistance that a tool can provide is therefore closely tied to the synthesis algorithm. Prior to the proliferation of AI-driven synthesis using Large Language Models (LLMs), most works that directly addressed validation challenges focused on program disambiguation [121, 202, 85], since their algorithms guaranteed that the set of solutions matched the user specification, but the ambiguity of the specification itself (input-output examples, types, etc.) left the user with the task of selecting the intended program among those matching the ambiguous specifications. LLMs, however, offer no such guarantees, and the current common specification, natural language, can be more ambiguous as well. So more recent work on LLMs explores validation more broadly, including the task of checking if any of the given programs match the specification at all. Many works explore the challenges in validating LLM-generated code [179, 110, 129, 190], and a few papers have suggested tools and interfaces for addressing these challenges [153, 180].

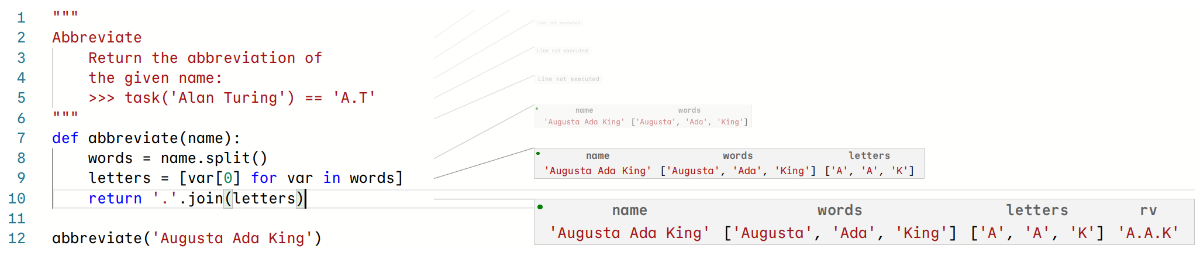
While validation is a necessary part of the interaction and appears in most chapters in this dissertation, it is a core contribution only in the last two chapters, focusing on interfaces for validating AI-generated code in a live environment. LEAP (Chapter 5) focuses on experienced developers in a LP-powered IDE, while ColDeco (Chapter 6) explores validation in the context of end user spreadsheet programmers.

1.1.2 Live Programming

A core interaction model to most chapters of this dissertation is *live programming*, so it is worth introducing it here. I will briefly cover what live programming is, walk through a short example of Projection Boxes, the live environment used most often here, and discuss the concept of *liveness* and what distinguishes a live environment from other interfaces such as read-eval-print-loops (REPLs) and computational notebooks.

Live programming[4], refers to a set of closely related programming paradigms involving the continuous execution of a program as it is being modified and extended [175]. A key motiva-

[4]Not to be confused with the practice of writing code live as a part of teaching a course [139].



```
1  """
2  Abbreviate
3      Return the abbreviation of
4      the given name:
5      >>> task('Alan Turing') == 'A.T'
6  """
7  def abbreviate(name):
8      words = name.split()
9      letters = [var[0] for var in words]
10     return '.'.join(letters)
11
12 abbreviate('Augusta Ada King')
```

```
Line not executed
Line not executed
Line not executed
        name                words
'Augusta Ada King' ['Augusta', 'Ada', 'King']
        name                words                  letters
'Augusta Ada King' ['Augusta', 'Ada', 'King'] ['A', 'A', 'K']
        name                words                  letters          rv
'Augusta Ada King' ['Augusta', 'Ada', 'King'] ['A', 'A', 'K'] 'A.A.K'
```

Figure 1.2. An example screenshot of PROJECTION BOXES [105], the Live Programming environment used in SNIPPY, LOOPY and LEAP.

tion for LP literature was and remains pedagogy [174, 74, 168, 91], but it has since extended to other areas. These include using LP for creative performance (also called *Live Coding*) [33, 20] and, more relevant to this dissertation, general purpose programming [105, 131, 95, 111, 90].

In the context of this dissertation then, LP refers to a programming paradigm where the dynamic state of the program is visualized during development, showing the runtime behavior of incomplete code, and updating automatically as the code is modified and extended. Note that this visualization may be as simple as the text representing each object [131], or as involved as a set of bespoke specialized visualizations [79].

PROJECTION BOXES. This dissertation uses PROJECTION BOXES [105], shown in Fig. 1.2, to explore interactive synthesis. PROJECTION BOXES is an LP environment for Python, built as a custom version of Visual Studio Code[5], which displays the available variables at each line of the program and updates automatically, triggered by user edits to the code.

In the example in Fig. 1.2, the code on the left is a simple Python program defining the function `abbreviate`, and calling it with a single example on line 12. PROJECTION BOXES executes this code automatically in the background, and displays the boxes on the right, containing the variables at each line. The box at line 8 includes an additional special variable `rv` denoting the return value of the function. Note that any changes to this code would result in the boxes being updated automatically to reflect the latest version of the code.

Liveness. To discuss what distinguishes live programming environments from other environ-

[5]https://code.visualstudio.com

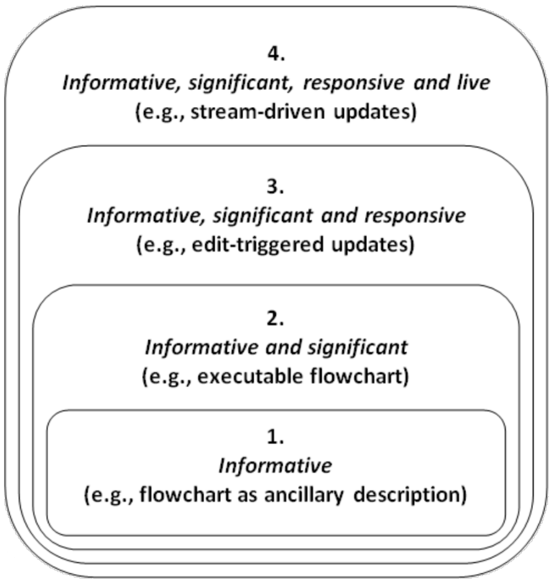
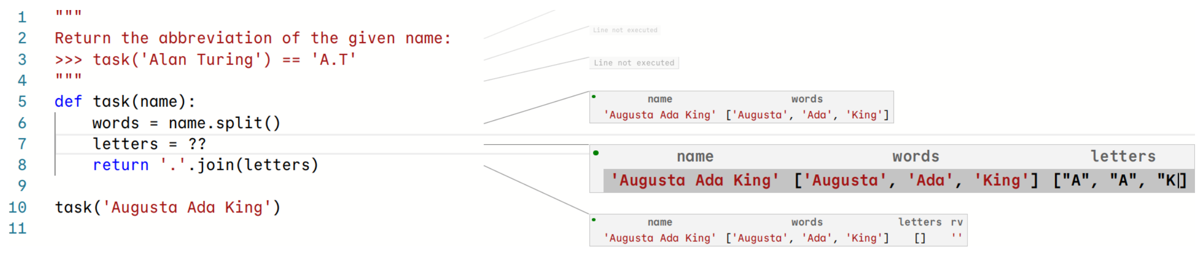

4.
Informative, significant, responsive and live
(e.g., stream-driven updates)
3.
Informative, significant and responsive
(e.g., edit-triggered updates)
2.
Informative and significant
(e.g., executable flowchart)
1.
Informative
(e.g., flowchart as ancillary description)

Figure 1.3. Four levels of liveness presented by Tanimoto [175].

ments that offer some form of execution feedback (*e.g.*, REPLs), *levels of liveness* are a useful concept. Originally introduced to discuss visual programming systems [174], they have since been updated [175] to include other programming environments, which you can see in Fig. 1.3. In this form, they present levels of program visualization based on how tied they are to the latest version of the program. At level 1, there is no connection (*e.g.*, the visualization is created manually), level 2 ties it directly to the source code, at level 3 the visualization responds to changes in the code automatically, while level 4 is fully live, responding to all changes, *e.g.*, in the file system or wall clock.

I would like to argue that these “levels”, rather than discrete categories, mark distinct points on a continuous scale. For instance, debuggers for compiled languages are clearly at level 2, tied to the program, but requiring explicit action from the user to run and visualize the latest runtime information. PROJECTION BOXES, meanwhile, are at level 3, responsive to edits to the code without requiring a separate action from the user. But where would REPLs fall in this categorization? They are responsive, but only to specific limited operations (usually some form of pressing “Enter” on the keyboard) and only display the final value of the last expression, not the intermediate values or side effects and changes to the state.



```
1   """
2   Return the abbreviation of the given name:
3   >>> task('Alan Turing') == 'A.T'
4   """
5   def task(name):
6       words = name.split()
7       letters = ??
8       return '.'.join(letters)
9
10  task('Augusta Ada King')
11
```

```
Line not executed

Line not executed

       name                     words
'Augusta Ada King' ['Augusta', 'Ada', 'King']

       name                     words                  letters
'Augusta Ada King' ['Augusta', 'Ada', 'King'] ["A", "A", "K|]

       name                     words           letters rv
'Augusta Ada King' ['Augusta', 'Ada', 'King']    []     ''
```

Figure 1.4. A screenshot of SNIPPY, showing the user providing the specification for the `letters` variable on line 7.

So I argue that instead of discrete categories, these levels present a spectrum of difference in (1) time and (2) required effort from the user towards seeing the latest visualization of program state. Then, the move from debuggers to live programming can be seen as a gradual move up the levels of liveness which improves various aspects of programming including code comprehension [26, 36] and debugging [5]. Thus, many of the findings discussed in this dissertation will apply to varying degrees to environments with at least some degree of liveness, such as REPLs and notebooks, but ultimately they are an exploration of what happens as we climb to higher levels of liveness.

1.2 Overview

With the background in program synthesis and live programming established, this section will briefly discuss each chapter in this dissertation, including some of their motivations and contributions.

SNIPPY*: Small-Step Live Programming by Example.* This first chapter takes two existing techniques, bottom-up enumerative program synthesis from input-output examples and live programming, and combines them into a new *small-step live PBE* interaction in our tool SNIPPY (Fig. 1.4). By leveraging LP, we were able to alleviate the need for manually providing long and tedious examples to the synthesizer, instead taking the *input*s from the LP environment directly and asking the user to merely specify their desired outputs. And by keeping the synthesis problem at the level of a single variable assignment, we could keep the synthesis runtime very short and so keep the interactive pace of development without disruptive interruptions. However, during the user study for SNIPPY, we quickly ran into two key limitations.

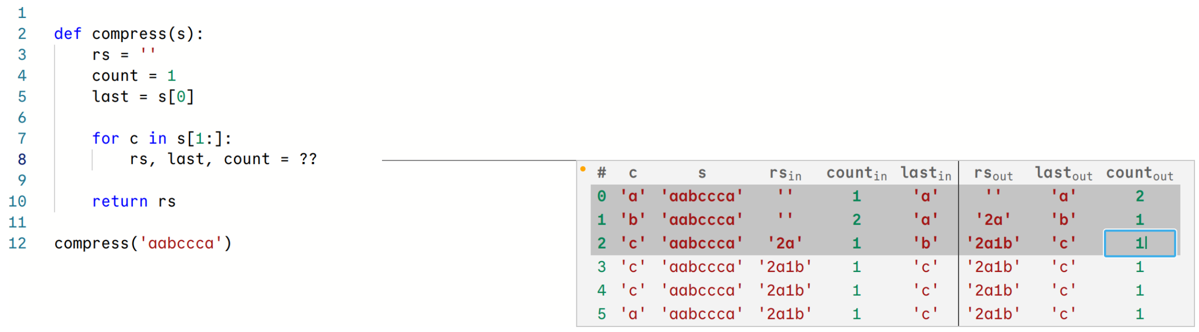
The first, which we dubbed the *user-synthesizer gap*, related to communicating the limitations of the synthesizer to the user. SNIPPY could fail for a number of reasons (invalid specifications, timeouts, and lacking the necessary components) but it had no way to communicate it meaningfully to the user. So users could get quickly discouraged and under-rely on the tool. On the other hand, some users with initial successes over-relied on SNIPPY, spending far more time trying to get SNIPPY to solve tasks that they may have been able to do themselves much more quickly. I will return to the issue of over- and under-reliance on synthesis in LEAP.

The second was a limitation in SNIPPY's algorithm and interface. Python users write imperative code, and want to invoke SNIPPY inside data-dependent loops and expect it to work well within (and ideally even synthesize) conditionals. But, following the SyGuS competition's benchmarks [9], we had developed SNIPPY to work in functional contexts with no side-effects and no control structures, instead allowing for `map` and `filter` operations using Python's list and dictionary comprehension. This was frustrating for our users at best, and resulted in incorrect synthesis results as worst. Addressing this is the topic of the next chapter.

LOOPY*: Bottom-up Enumerative Synthesis with Control Structures.* LOOPY was a direct result of our user study for SNIPPY. By observing users, how they wrote imperative Python code, and where and how they invoked the synthesizer, we were able to extend SNIPPY and its core algorithm to better meet our users' needs in LOOPY. The core of LOOPY's contribution is *Block-level live PBE*, where the user is able to provide the specification for entire blocks of code rather than a single variable assignment, and LOOPY can correctly and quickly synthesize the desired programs in the presence of control structures.

To do this, we needed to extend both the user interface and the algorithm simultaneously[6]. To allow for synthesis inside data-dependent loops (where future iterations of the loop

[6]See Peleg [140] for a discussion of this philosophy of design and development, dubbed "Co-Design", including a deeper discussion of LOOPY's development.



```
1
2   def compress(s):
3       rs = ''
4       count = 1
5       last = s[0]
6
7       for c in s[1:]:
8           rs, last, count = ??
9
10      return rs
11
12  compress('aabccca')
```

#	c	s	rs_{in}	$count_{in}$	$last_{in}$	rs_{out}	$last_{out}$	$count_{out}$
0	'a'	'aabccca'	''	1	'a'	''	'a'	2
1	'b'	'aabccca'	''	2	'a'	'2a'	'b'	1
2	'c'	'aabccca'	'2a'	1	'b'	'2a1b'	'c'	1
3	'c'	'aabccca'	'2a1b'	1	'c'	'2a1b'	'c'	1
4	'c'	'aabccca'	'2a1b'	1	'c'	'2a1b'	'c'	1
5	'a'	'aabccca'	'2a1b'	1	'c'	'2a1b'	'c'	1

Figure 1.5. Providing a specification inside a data-dependent loop in LOOPY. Notice that while the code on line 8 is yet to be synthesized, the input values in the PROJECTION BOXES are correct up to the third iteration.

depend on the execution of as-yet-unsynthesized code from previous iterations) and conditionals, we developed *live execution*, stepping over the hole in the body left for synthesis, and using the user's specified output values to fill-in that hole and correctly execute future iterations. You can see an example of this in Fig. 1.5.

To synthesize entire blocks of variable assignments, we developed the *Intermediate State Graph* (ISG), a hypergraph compactly representing all possible orders of assignments and an optional top-level conditional over the entire block. The details of the ISG and algorithm are beyond the scope of this introduction. But, in short, aside from the before-state (the state of the program before the synthesized code) and after-state (the state after the execution of the to-be-synthesized code), each variable assignment could be seen as a *mutation* of the state, resulting in a number of intermediate states. Since LOOPY is restricted to one mutation per variable, we could efficiently enumerate all possible orders of variable assignments by representing each of these states as nodes in a graph, with edges connecting two nodes if a variable assignment mutated the parent node, and resulted in the child node. This restricted form of mutation worked well for LOOPY, allowing it to synthesize blocks of code in just a few seconds. But it also hinted at a deeper limitation of bottom-up enumerative synthesis algorithms: their inability to easily synthesize mutating code, which is the topic of the following chapter.

SOBEQ*: Bottom-up Synthesis of Side-Effects with Separation Logic.* We had chosen bottom-

$$\textsc{Frame}\ \dfrac{\{\mathtt{n}\mapsto 5\}\mathtt{n}\{\mathtt{n}\mapsto 5;5\}}{\{\mathtt{arr}\mapsto \mathtt{[]} * \mathtt{n}\mapsto 5\}\mathtt{n}\{\mathtt{arr}\mapsto \mathtt{[]} * \mathtt{n}\mapsto 5;5\}}$$

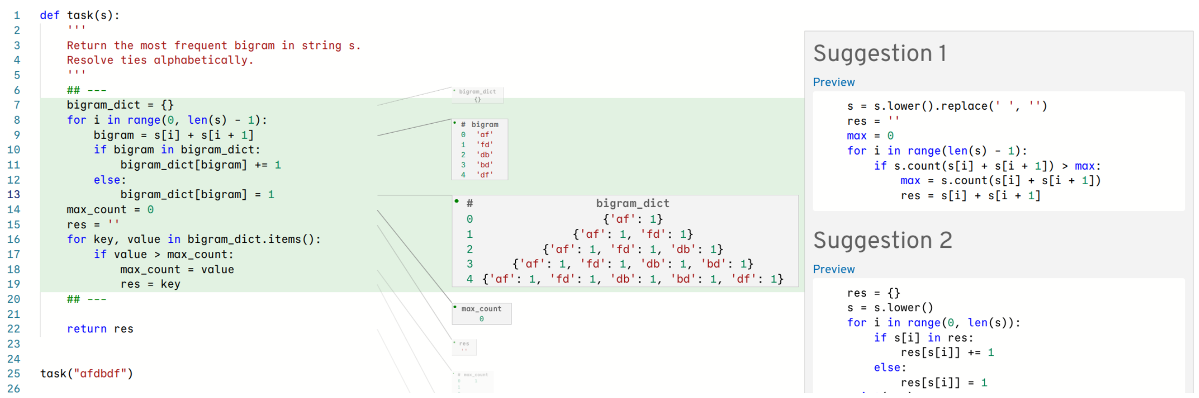
$$\textsc{Eval}\ \dfrac{\textsc{Frame}\ \dfrac{\textsc{Eval}\ \dfrac{\{\mathtt{arr}\mapsto \mathtt{[1]}\}\mathtt{arr}\{\mathtt{arr}\mapsto \mathtt{[1]};\mathtt{[1]}\}}{\{\mathtt{arr}\mapsto \mathtt{[1]}\}\mathtt{arr.pop()}\{\mathtt{arr}\mapsto \mathtt{[]};1\}}}{\{\mathtt{arr}\mapsto \mathtt{[1]} * \mathtt{n}\mapsto 5\}\mathtt{arr.pop()}\{\mathtt{arr}\mapsto \mathtt{[]} * \mathtt{n}\mapsto 5;1\}} \qquad \downarrow}{\{\mathtt{arr}\mapsto \mathtt{[1]} * \mathtt{n}\mapsto 5\}\mathtt{arr.pop()\ +\ n}\{\mathtt{arr}\mapsto \mathtt{[]} * \mathtt{n}\mapsto 5;6\}}$$

Figure 1.6. The CHSL derivation of the program `arr.pop() + n` in SOBEQ.

up enumeration as the synthesis algorithm for SNIPPY and LOOPY because, in centering the user's experience of our tools, we needed an algorithm that would work with simple input-output examples, but could synthesize solutions quickly and with a deterministic runtime.

To do this synthesis efficiently, however, this algorithm relies on *Observational Equivalence* (OE) reduction [6, 177] a technique that allows it to prune large parts of its search space by discarding programs that, on the given before-state, produce the same output value (*i.e.*, are "observationally" equivalent). This causes problems if the synthesized programs can mutate that before-state in any way, not only potentially losing the desired solution by discarding a program that appears equivalent in the before-state but not after some synthesized mutation, but possibly synthesizing an incorrect program by not accounting for mid-expression mutations. We bypassed this problem in LOOPY by restricting permitted mutations to a fixed set of possible variable assignments, and enumerating programs for each intermediate state separately. But allowing arbitrary mutations was an open problem.

This chapter introduces a novel bottom-up enumerative algorithm that can correctly and efficiently synthesize programs with mutations. To do so, we used a simple Separation Logic [151, 137] we named *Concrete Heap Separation Logic* (CHSL), representing the operational semantics of the synthesized language. Instead of enumerating all programs in a fixed before-state, we instead enumerate hoare-triples including the pre- and post-condition of each enumerated program, and redefine the synthesis goal as a derivation in CHSL of the desired program. You can see a simple example of such a derivation in Fig. 1.6.



```
def task(s):
    '''
    Return the most frequent bigram in string s.
    Resolve ties alphabetically.
    '''
    ## ---
    bigram_dict = {}
    for i in range(0, len(s) - 1):
        bigram = s[i] + s[i + 1]
        if bigram in bigram_dict:
            bigram_dict[bigram] += 1
        else:
            bigram_dict[bigram] = 1
    max_count = 0
    res = ''
    for key, value in bigram_dict.items():
        if value > max_count:
            max_count = value
            res = key
    ## ---

    return res


task("afdbdf")
```

```
bigram_dict
    {}
```

```
# bigram
0  'af'
1  'fd'
2  'db'
3  'bd'
4  'df'
```

```
#                  bigram_dict
0                   {'af': 1}
1              {'af': 1, 'fd': 1}
2          {'af': 1, 'fd': 1, 'db': 1}
3      {'af': 1, 'fd': 1, 'db': 1, 'bd': 1}
4 {'af': 1, 'fd': 1, 'db': 1, 'bd': 1, 'df': 1}
```

```
max_count
    0
```

```
res
 ''
```

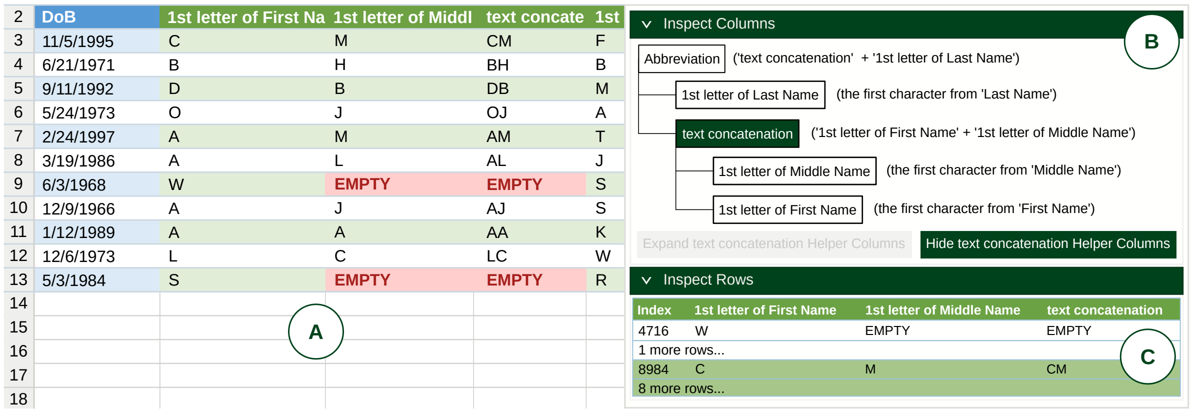
```
s = s.lower().replace(' ', '')
res = ''
max = 0
for i in range(len(s) - 1):
    if s.count(s[i] + s[i + 1]) > max:
        max = s.count(s[i] + s[i + 1])
        res = s[i] + s[i + 1]
```

```
res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
```

Figure 1.7. An example screenshot of LEAP. The suggestion panel on the right is inspired by Copilot's multi-suggestion pane, but rather than a single-use `accept solution` button, it allows the user to preview each suggestion, allowing them to take advantage of static analysis results and live programming.

The above three chapters explore interaction models and algorithms for PBE but, more recently, LLM-driven code generation has shifted the focus of human-centered program synthesis research. LLMs can generate orders of magnitude more code within the same time as PBE and other PL-based algorithms. And, since they work with natural language and code context, they incur an even lower specification overhead than PBE, acting as a significantly more powerful autocomplete [60], and in theory allowing those with no programming experience to write code [112]. LLM-driven synthesis comes with its own limitations, however, with one of the most vital open problems being that of *validation*. LLMs offer no guarantees about the generated code (not even syntactic validity), and so burden the user with the work of validating that the code is syntactically correct, semantically meaningful, and matches the user's intent. The last two chapters of this dissertation focus on this problem of validating AI-generated programs, first in a similar environment as SNIPPY and LOOPY, but now powered with LLMs and capable of producing multiple solutions, and finally in spreadsheets, arguably the most widely used live programming environment.

LEAP*: Live Exploration of AI-Generated Programs.* Our work on LEAP was greatly inspired by Grounded Copilot [13]. In their study of how developers use Copilot, they found that users tend to use the tool in one of two modes: *Acceleration* where the tool assists them in completing



2	DoB	1st letter of First Na	1st letter of Middl	text concate	1st
3	11/5/1995	C	M	CM	F
4	6/21/1971	B	H	BH	B
5	9/11/1992	D	B	DB	M
6	5/24/1973	O	J	OJ	A
7	2/24/1997	A	M	AM	T
8	3/19/1986	A	L	AL	J
9	6/3/1968	W	EMPTY	EMPTY	S
10	12/9/1966	A	J	AJ	S
11	1/12/1989	A	A	AA	K
12	12/6/1973	L	C	LC	W
13	5/3/1984	S	EMPTY	EMPTY	R
14					
15					
16					
17					
18					

Inspect Columns

Abbreviation ('text concatenation' + '1st letter of Last Name')
1st letter of Last Name (the first character from 'Last Name')
text concatenation ('1st letter of First Name' + '1st letter of Middle Name')
1st letter of Middle Name (the first character from 'Middle Name')
1st letter of First Name (the first character from 'First Name')

Inspect Rows

Index	1st letter of First Name	1st letter of Middle Name	text concatenation
4716	W	EMPTY	EMPTY
1 more rows...			
8984	C	M	CM
8 more rows...			

A
B
C
Expand text concatenation Helper Columns
Hide text concatenation Helper Columns

Figure 1.8. An example screenshot of COLDECO, showing decomposed columns (A), natural language descriptions (B), and summary rows (C).

a known solution faster, and *Exploration* where the tool assists them in exploring potential solutions to their problem. Their findings suggested that users spend significantly more time in exploration mode, and that in this mode, they are much more willing to explore alternative solutions, use the set of affordances at their disposal (direct examination, code execution, etc.) to validate the suggestions, and edit or even cherry-pick from multiple suggestions.

With these findings in mind, we developed LEAP, a set of improvements to Copilot's *multi-suggestion pane* combined with live programming in the form of PROJECTION BOXES, which could significantly lower the cost of validation. In a user study, we found that this lower cost decreased both over- and under-reliance on generated code, and resulted in an overall lower cognitive load. Though these benefits were highly dependent on the type of task, with one-off and API-heavy tasks benefitting more than general algorithmic tasks.

COLDECO: An End User Spreadsheet Inspection Tool for AI-Generated Code. All previous chapters focus on users with at least some degree of programming experience. But the first major success of program synthesis, and one of its most promising future directions, is its ability to empower *end user programmers*, *i.e.*, those without programming experience, to work with and write code. So the final chapter of this dissertation shifts the focus away from the IDE, and turns to the problem of synthesized program validation in spreadsheets.

To this end, we developed three interfaces for end user validation of AI-generated programs, and combined them in our tool ColDeco (see Fig. 1.8). First, inspired by the work of Liu et al. [112], we provided a *natural language description* of the generated code, using a fully deterministic template-based system. Next, we gave the users the ability to *decompose* the single output column created by the synthesized code into "helper" columns displaying intermediate values. Finally, we showed users *summary rows*, a set of rows selected from the table demonstrating the different behaviors of the program. Crucially, ColDeco did not show the generated Python code to the users. Our study found that users could successfully validate AI-generated programs using ColDeco, and that they had a diversity of preferences for the affordances ColDeco provides.

Chapter 2

SNIPPY: Small-Step Live Programming by Example

2.1 Introduction

Live programming is a paradigm where the programming environment continually displays runtime values. While live programming provides immediate feedback about the current state of execution, it does not explicitly help the programmer to discover the next line of code they need to write to accomplish their goal.

On the other hand, program synthesis is a technique that helps programmers by generating code automatically. There are many approaches to program synthesis, but in this paper we focus on a class of techniques called Programming-by-Example (PBE), where the programmer provides input-output examples, and the synthesizer produces candidate programs that satisfy these examples. While program synthesis can generate code that accomplishes a given goal, traditional synthesizers are stand-alone and not integrated tightly into the development work-flow, which makes it hard for the programmer to formulate the goal for the synthesizer to solve.

As such, live programming and PBE are perfectly suited for each other: the live programming environment provides all the values needed for the programmer to easily provide examples without a significant break in workflow; and synthesis from examples helps address a limitation of live programming, which is that it does not explicitly generate statements.

Because of this symbiotic relationship, there has been prior work on combining Live

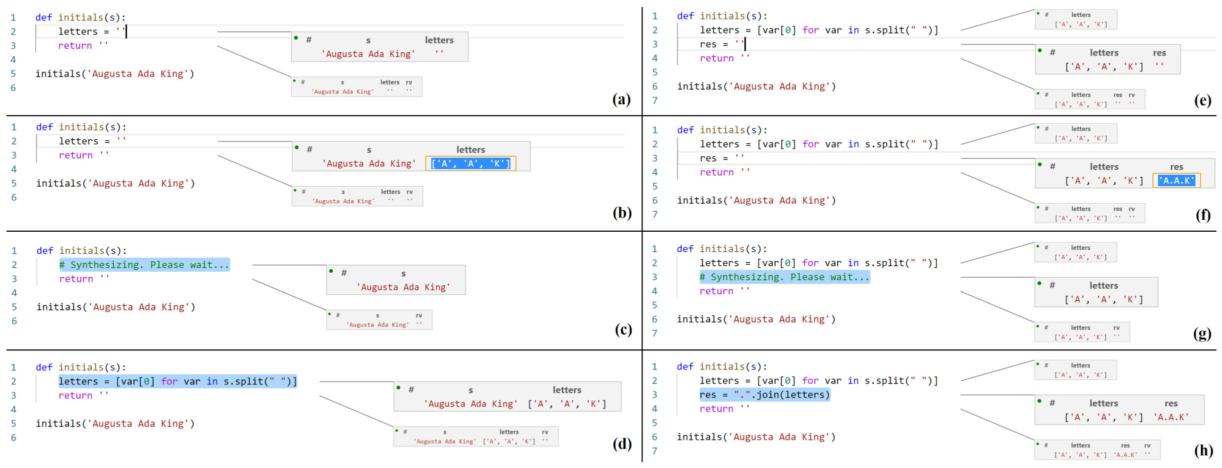
Programming with PBE, sometimes called Live Programming by Example [159, 154] or synthesis from Direct Manipulation Interfaces [32, 120, 77]. However, broadly speaking, examples in this prior work describe behavior holistically, meaning that each example impacts either the entire program, or a large part of the program (e.g., an entire function). In the literature on program semantics, this kind of specification is usually referred to as a big-step semantics [193].

In this paper, we describe a different approach to Live Programming by Example, which we call *Small-Step Live Programming by Example* (SSL-PBE). In contrast to prior work, SSL-PBE allows the programmer to specify examples in a live programming environment, but only for a single missing statement. Synthesis in SSL-PBE starts in a live programming environment where program state is displayed after each statement. While in traditional live programming the displayed state is read-only, in SSL-PBE the runtime values can be modified. When the programmer edits values in the state, a program synthesizer runs to generate a local program snippet that satisfies the new data. SSL-PBE is unique in that it enables a new programming paradigm where the programmer "leads" the generation of the program with data.

To understand the viability of this new paradigm, we implemented SSL-PBE for the Python programming language, in a tool named SnipPy. SnipPy uses the live programming environment of Projection Boxes [105] and a custom-made enumerative program synthesizer to generate Python statements.

Through a user study, we demonstrate that SSL-PBE is easy to use, and has an impact on task time and correctness on more difficult tasks. Our study also shows that the synthesizer can generate between 18% to 66% of the code, thus demonstrating that the synthesizer and the human can work together to form a complete solution. Finally, our user study also shows that almost all our participants preferred SnipPy over searching the internet in some cases, the main reasons being that compared to the internet searches SnipPy incurs a lower cognitive burden, automatically connects the snippet with the surrounding code, and provides a more compact solution.

The main contributions of this paper are:


(a)
```
def initials(s):
    letters = ''
    return ''

initials('Augusta Ada King')
```
```
#  s                   letters
   'Augusta Ada King'  ''
#  s                   letters  rv
   'Augusta Ada King'  ''       ''
```
(b)
```
def initials(s):
    letters = ''
    return ''

initials('Augusta Ada King')
```
```
#  s                   letters
   'Augusta Ada King'  ['A', 'A', 'K']
#  s                   letters  rv
   'Augusta Ada King'  ''       ''
```
(c)
```
def initials(s):
    # Synthesizing. Please wait...
    return ''

initials('Augusta Ada King')
```
```
#  s
   'Augusta Ada King'
#  s                   rv
   'Augusta Ada King'  ''
```
(d)
```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    return ''

initials('Augusta Ada King')
```
```
#  s                   letters
   'Augusta Ada King'  ['A', 'A', 'K']
#  s                   letters          rv
   'Augusta Ada King'  ['A', 'A', 'K']  ''
```
(e)
```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ''
    return ''

initials('Augusta Ada King')
```
```
#  letters
   ['A', 'A', 'K']
#  letters          res
   ['A', 'A', 'K']  ''
#  letters          res  rv
   ['A', 'A', 'K']  ''   ''
```
(f)
```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ''
    return ''

initials('Augusta Ada King')
```
```
#  letters
   ['A', 'A', 'K']
#  letters          res
   ['A', 'A', 'K']  'A.A.K'
#  letters          res  rv
   ['A', 'A', 'K']  ''   ''
```
(g)
```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    # Synthesizing. Please wait...
    return ''

initials('Augusta Ada King')
```
```
#  letters
   ['A', 'A', 'K']
#  letters
   ['A', 'A', 'K']
#  letters          rv
   ['A', 'A', 'K']  ''
```
(h)
```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ".".join(letters)
    return ''

initials('Augusta Ada King')
```
```
#  letters
   ['A', 'A', 'K']
#  letters          res
   ['A', 'A', 'K']  'A.A.K'
#  letters          res      rv
   ['A', 'A', 'K']  'A.A.K'  ''
```

Figure 2.1. Writing code using SNIPPY: (a)-(d) generates the first statement and (e)-(h) generates the second statement.

- We present a novel paradigm called Small-Step Live Programming by Example, in which programmers can modify live data to generate code snippets.
- We present an implementation of this paradigm in a tool called SNIPPY.
- A user study of SNIPPY on 13 programmers finding that SNIPPY complements web searches in bridging certain types of knowledge gaps, and that SNIPPY helped users solve harder problems
- We identify the *user-synthesizer gap*, where a mismatch between the user's mental model of the synthesizer and the abilities of the synthesizer hinders the user's ability to use the synthesizer effectively. We believe that the *user-synthesizer gap* needs to be addressed as synthesis tools begin to target programmers rather than end-users.

2.2 Motivating example

A programmer named Kayla is processing a text file using Python. One part of the processing involves reducing a name (e.g., "Augusta Ada King") to a non-standard form of initials[1] (e.g., "A.A.K"). Kayla has a lot of programming experience, but is only a casual Python

[1]The task is taken from this competitive programming exercise: https://www.codewars.com/kata/57eadb7ecd143f4c9c0000a3

user, which means Kayla does not immediately know how to achieve this task in Python. Being an experienced programmer, Kayla breaks down the task into two components: getting the first letter of each word, and reconnecting them in the desired format.

2.2.1 Opportunistic programming

Kayla turns to searching for the answer using a search engine. Unsure of the precise string terminology of Python, but used to relying on search engines, she tries the natural language search, “Python first letter of every word”. The first result is a link to the Stack Overflow question “How can I get the first letter of each word in a string?” which has no accepted answer but upon further reading has code which looks suitable in the comments discussing the question. Copying the code and modifying the variable names, Kayla now has the following:

```
letters = [w[0] for w in s.split(' ')]
```

This code returns a list of initials. Now all that remains is to format it. Since the required formatting is trickier than standard initials punctuation, Kayla knows that a loop is not the easiest way to go in this case. Kayla recalls that Python has a `join` method to convert arrays to strings, so she tries:

```
letters = [w[0] for w in s.split(' ')]
res = letters.join('.')
```

However, this code produces the runtime error

```
AttributeError: 'list' object has no attribute 'join'
```

which Kayla now looks up online as well. After some digging, Kayla realizes that `join` is a method on strings, so the correct code is in fact:

```
letters = [w[0] for w in string.split(' ')]
res = '.'.join(letters)
```

2.2.2 Small-Step Live Programming by Example

Let us consider the same task again, but instead Kayla will use our proposed approach, Small-Step Live Programming by Example, as reified in our SNIPPY tool. Kayla starts in a live programming environment, in this case Projection Boxes [105], as seen in Fig. 2.1(a). The visualization shows at each line a projection box with the values of all variables at that line. However, while in a traditional live programming environment the visualization is read-only, in our approach the values of variables in the visualization can be edited to show the programmer's intent. Thus, Kayla edits the value of `letters` in the projection box to enter the desired value, as shown in Fig. 2.1(b). By this, Kayla is stating that she wants SNIPPY to generate a code snippet that will produce `['A','A','K']` in the `letters` variable when `s` is `'Augusta Ada King'`. In the synthesis literature this is called an *input-output example*. If the statement Kayla wanted to generate was executed multiple times (i.e., it is inside a loop or in a function that is called multiple times), the projection box would have one line per execution and Kayla would be able to provide one or more input-output examples, one for each execution of the statement.

Once Kayla has provided some input-output examples by changing the live visualization, these examples are sent to a Programming by Example (PBE) synthesis engine, while the user is told that the synthesizer is working in the background Fig. 2.1(c). In previous PBE tools aimed at programmers, providing examples is often a weak point of the interaction model, sometimes requiring a break in the workflow that could be as severe as switching to a different tool and editing its configuration files. Through direct manipulation of live data, our approach makes the specification process seamless. Furthermore, focusing the user's attention on the value assigned to a single variable turns the synthesizer into a helper utility in a larger task, which harnesses the still-limited power of synthesis to solve specific sub-tasks for the user.

Within seconds, the SNIPPY synthesizer finds a solution and adds the generated code snippet, as shown in Fig. 2.1(d).

Next, Kayla creates a new variable, called `res`, as shown in Fig. 2.1(e) – for brevity of the figure, at this point we configured Projection Boxes to not display the variable `s` anymore. Kayla changes the projection box value of `res` to be `'A.A.K'`, as shown in Fig. 2.1(f). This provides the synthesizer with an input-output example stating the output in `res` should be `'A.A.K'` when `s` is `'Augusta Ada King'` and `letters` is `['A', 'A', 'K']`. Within a few seconds, the SNIPPY synthesizer generates a statement, as shown in Fig. 2.1(g)-(h). To finish the code, Kayla changes the return statement to return the `res` variable.

In summary, this example showed how Kayla was able to finish writing the code, without leaving the IDE, without searching online, and without bearing the cognitive load of thinking about the details of list comprehension, `split` or `join`.

2.3 Related work

2.3.1 Live programming

Live programming is a paradigm that provides immediate visualizations of a program's behavior. This research area dates back to the seminal work of Hancock [74]. Live programming environments have been developed for Python [68, 90], Java [18], Javascript [149, 1], Lisp [2] and ML-like languages [132]. There is also work categorizing different kinds of liveness [175, 183], and studies exploring the benefits of live programming [191, 95]. Traditionally, live programming only provides feedback to the programmer, and not the ability to edit the output or synthesize new pieces of code. Our work distinguishes itself from traditional live programming by contributing a new paradigm where changing data in the visualization can produce small code snippets for the programmer.

2.3.2 Big-step program synthesis and repair

Program synthesis takes a specification of behavior and returns a program satisfying the specification. Automatic program repair (APR) takes a program and a specification, and minimally changes the program to satisfy the specification. Both commonly take specifications that describe the full program, or as previously desribed, “big-step” specifications. Most synthesis and APR literature assumes a user that provides the specifications, but does not focus on the experience of the user when creating the specifications or processing the results.

Programming by Example

Programming by Example (PBE) is a field of program synthesis where behavior specifications are provided as input-output pairs. This approach has been applied to string transformations of Excel data [63] and file renaming [66], data extraction [100], and transforming tabular data [195], all synthesis solutions for end-users. In APR and synthesis aimed at developers, examples can also take the form of unit tests [115, 116, 194, 101].

Since examples are partial specifications, often resulting in too-specific programs that require more examples to generalize, [143] question their sufficiency as a specification tool. Entering examples is also error-prone, a problem tackled by [142].

Live programming by example

Two approaches for synthesis for developers are named “Live Programming by Example”. The first [159] allows users to edit HTML output causing changes to its rendering JavaScript code. The second [154] consists of two connected files, a code file with functions and an examples file with unit tests for the functions. Unit tests are executed for live programming data, and modifying a unit test will attempt to synthesize a new function where the tests pass. Both approaches shift the audience of PBE from non-programmer end-users to programmers who interact with the synthesized code, but both still use examples as a holistic mode of specification. In SNIPPY, the separation between code and data is not as severe: live values are adjacent to the code, not in another file. Additionally, SNIPPY uses examples as small-steps on intermediate values, rather than just the output of the entire execution, giving users access to localized synthesis from local examples.

Domain specific synthesis tools

Several recent projects focus on *domain-specific* example-driven tools for non-programmers: Rousillon [29] generates web scrapers, Bespoke [178] GUI applications, and Wrex [41] Python wrangling code in Jupyter Notebooks. SNIPPY targets general programming, rather than one domain-specific task. As such, our intended users are programmers, not data scientists or power-users, and we augment a different workflow.

Direct manipulation interfaces as specification

Direct manipulation user interfaces [162] are visualizations of the system state that can be changed, and are prevalent as editors of graphical representations. Sketch-n-Sketch [32, 120, 77] is a direct-manipulation editor for *programmable* graphical formats such as SVG, HTML, and LaTeX, where direct manipulation changes to the visual output become repair operations on the generating code and inputs. [77] takes this further by adding refactoring tools and turning the direct manipulation editor into a visual programming environment. The direct manipulation workflow is aimed at modifying visual objects in languages with little separation between code and input. SSL-PBE is a technique for more expressive langauges.

2.3.3 Small-Step Program Synthesis

Several existing synthesis and APR techniques rely on a small-step specification, or specifications that tackle only local behavior within a larger program.

Sketch and sketching

Sketching (popularized by Sketch [167] and modified into a variant used by many synthesis works [57, 164, 81, 141]) is a method for specifying to the synthesizer a partial implementation of the target program with *holes*, initially missing numeric values but now missing expressions or statements. Sketches isolate local steps for the synthesizer to fill, and are usually accompanied by a big-step specification (e.g., examples) from which a local specification for the holes is derived.

SNIPPY differs from these in two ways: first, synthesis queries in SNIPPY are themselves locally-specified, and do not require the potentially-lossy transformation from a larger specification, and second, and enabled by the previous, SNIPPY can be used without end-to-end specifications for the full program, which means it can be used to perform exploration.

Single statement synthesis

Small-step synthesis has been worked into IDEs to attempt predicting the next statement. For example, InSynth [72] code-completes assignments using the type of the assigned variable as the specification. As another example, CodeHint [57] allows the programmer to stop the program at a breakpoint and generate a new version of the next statement by providing a specification on the current program state (typically a typef-based specification, though examples are possible). The user can trigger the breakpoint again to provide specifications for more inputs, but those only serve to rule out candidates generated for the first input state. While in CodeHint providing specifications for multiple inputs requires running to the breakpoint again and again, in SNIPPY the PROJECTION BOXES display flattens both loops and multiple inputs into a single list, allowing the user to specify any and all inputs at once.

Both CodeHint and InSynth return several ranked options for the completion, a different interaction model than that of SNIPPY, which returns a single result. While returning a single result is an all-or-nothing approach, it also means that users do not need to select out of multiple options, a task that previous work has shown is difficult for users to perform correctly without in-depth inspection [143].

Program states as specifications

Another form of local specifications is *direct state manipulation*, or changing the program's internal state to create repair specifications, as in Wolverine [182] and JDial [80]. Both provide ways for users to modify the values of variables along an execution trace of the program on a test input, then create a repair to the program. Wolverine, which has a gdb-like interface, also hot-patches the repair and allows debugging to continue.

Unlike both projects, which are APR tools, SNIPPY is a synthesis tool. This means that SNIPPY's user is not working on an already fully-written program and only correcting a bug found in the course of the execution.

Both projects also tackle a common problem: APR can find a trivial fix that satisfies a specification by removing desired functionality. Wolverine's solution is to let the programmer, while debugging the program, mark an intermediate state as specification, declaring the *unmodified state* should appear in the execution trace of the final program. SNIPPY faces a similar problem, declaring certain behavior as already-correct while not forcing the user to specify all inputs. Its solution, discussed in the next section, is inspired by Wolverine's solution, adapted from repair to synthesis.

2.4 User Interface

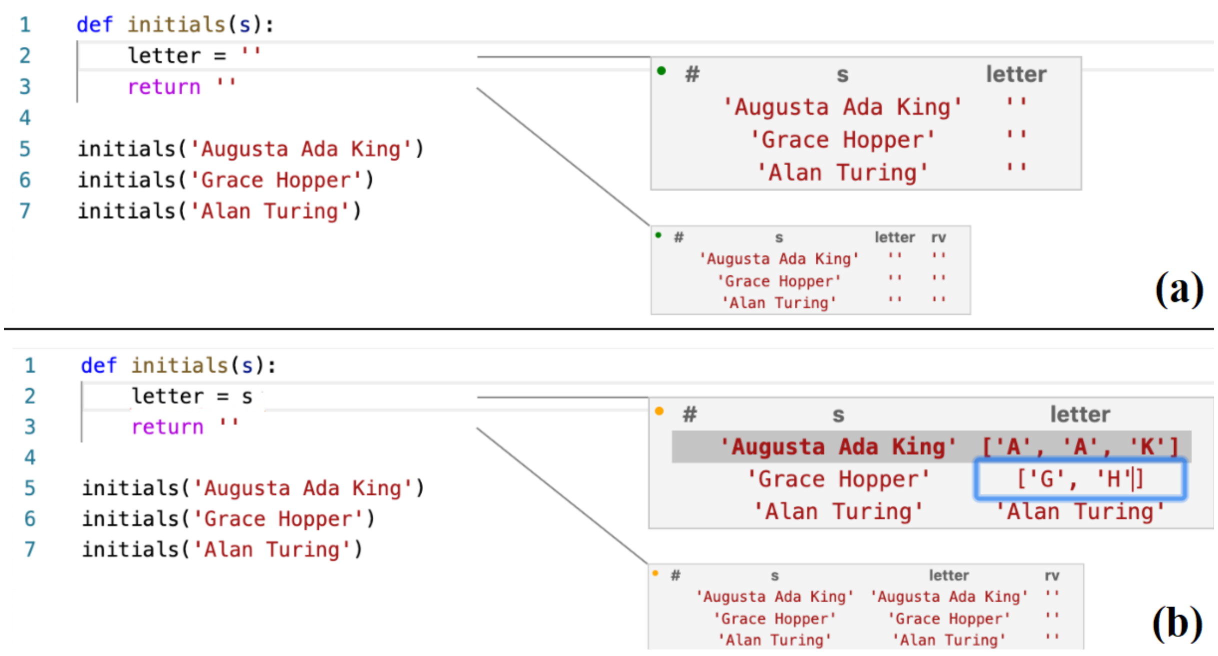
We implemented SNIPPY on top of PROJECTION BOXES [105] by making the variable values editable in each projection box. After the programmer modifies some values in a projection box, every modified row becomes an example that is sent to a Programming by Example (PBE) synthesizer. Recall that the projection box for a line in the program contains multiple rows if that line is executed multiple times.

Activating SNIPPY

In order to allow SNIPPY to naturally become part of the development workflow, it can be activated and specified entirely via keyboard operations. To assign a synthesized value to a variable, the user begins typing an assignment statement, but instead of a concrete value, they assign `??`:

```
newVar = ??
```

This special token, which is not valid Python code, temporarily becomes the temporary assigned value 0 (since Python variables are not typed, a specific initial value by type cannot be generated), and the focus is moved to the now-editable value of the variable, as in Fig. 2.1(b). The



```
def initials(s):
    letter = ''
    return ''

initials('Augusta Ada King')
initials('Grace Hopper')
initials('Alan Turing')
```

#	s	letter
	'Augusta Ada King'	''
	'Grace Hopper'	''
	'Alan Turing'	''

#	s	letter	rv
	'Augusta Ada King'	''	''
	'Grace Hopper'	''	''
	'Alan Turing'	''	''

```
def initials(s):
    letter = s
    return ''

initials('Augusta Ada King')
initials('Grace Hopper')
initials('Alan Turing')
```

#	s	letter
	'Augusta Ada King'	['A', 'A', 'K']
	'Grace Hopper'	['G', 'H']
	'Alan Turing'	'Alan Turing'

#	s	letter	rv
	'Augusta Ada King'	'Augusta Ada King'	''
	'Grace Hopper'	'Grace Hopper'	''
	'Alan Turing'	'Alan Turing'	''

Figure 2.2. Using SNIPPY with multiple values. (a) PROJECTION BOXES showing multiple values for the same line, and (b) providing only some of the values as examples to SNIPPY.

user can then enter a new value, and start the synthesizer by hitting Enter.

Modified values are then packaged as examples to the synthesizer, where each input state includes all variables that are in scope at the line that is being synthesized, namely all variables that appear in the projection box at that line.

Editing examples

Some examples are inconvenient to type, but easier to edit. For example, if the user wants to turn the string `'Augusta Ada King'` in the variable `s` into a list of words `['Augusta', 'Ada', 'King']`, they can start with a blank variable value and enter in every word into the list, but it is easier and far less error prone to start with the value of the string and edit it into a list. To this end, the user can activate SNIPPY with an expression reference:

```
newVar = s??
```

This will put the user in edit mode for the values of `newVar` as before, but instead of a default initial value will populate `newVar` with the value of `s`.

Multiple examples

PROJECTION BOXES can show multiple values for a line of code. This can happen in one of two cases: if the function is called multiple times with multiple values, as seen in Fig. 2.2(a), or if the current line of code is inside a loop. For SNIPPY, this means that there are different values for the in-scope variables that can be sent to the synthesizer as multiple examples for the same expression. When the PROJECTION BOXES contain multiple rows, the edited variable becomes editable in all rows, as seen in Fig. 2.2(b), and the user can travel between the values using the Tab key.

Sometimes the user does not want to provide an output value for every row in the PROJECTION BOXES. A simple example for this would be inside a loop with many iterations, where two or three examples will suffice to demonstrate the desired functionality. SNIPPY allows users to only edit some of the output values. Rows where the value was changed are submitted to the synthesizer as examples, and unchanged rows are ignored. Fig. 2.2(b) shows highlighted rows that were edited by the user and will be sent as examples to the synthesizer.

Occasionally, the value of some of the rows already exhibits correct behavior. These already-correct rows will not be changed, but if they are not sent to the synthesizer along with changed rows, a program that changes their behavior may be synthesized. To avoid this, SNIPPY lets the user specify that the output value in a row is correct as-is by selecting it with Shift+Enter. A variant of this problem was previously solved in Wolverine [182], where the user can mark a state as "specification" the execution must still pass through, but with no assurance which line of code will pass through it.

A Synchronous Modality

SNIPPY synthesis calls are synchronous. This means that once the user calls the synthesizer, a wait message (shown in Fig. 2.1(c)) appears, and the user waits for synthesis to quickly finish or fail. An asynchronous workflow could have been explored, but it would not allow SNIPPY to be used in a more exploratory manner. The synchronous workflow motivates us to

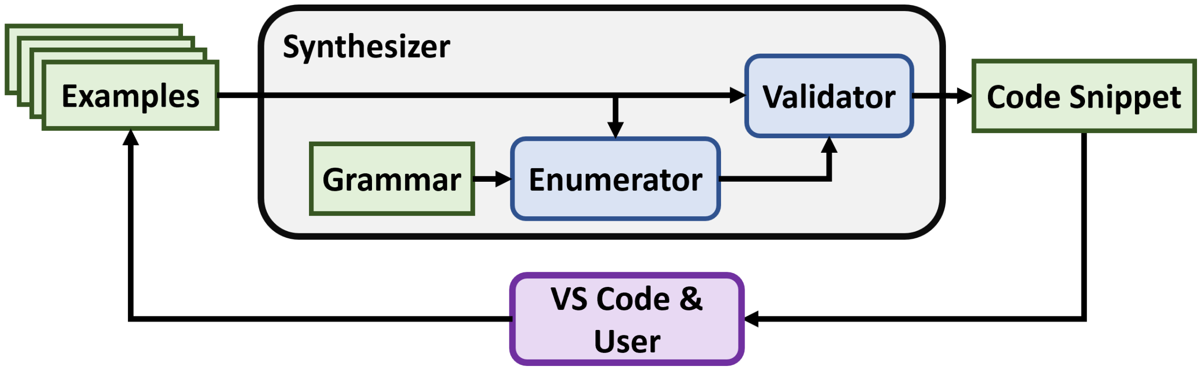


Figure 2.3. The structure of our *generate-and-test* synthesizer. Programs are enumerated and passed to the validator to be tested against the user-provided examples.

use a timeout that is as brief as possible, which means our synthesizer must be extremely efficient to find meaningful expressions within that timeout. We chose a timeout of seven seconds, enough to synthesize programs of up to height 3 (zero-based) on several desktop and laptop architectures, while staying well below a distruptive interruption [136]. In the next section, we discuss the design considerations when building such a synthesizer.

2.5 Synthesizer implementation

We designed and implemented a custom synthesizer to generate the one-line Python snippets requested by SNIPPY users. The synthesizer that we built is known as an *enumerating generate-and-test* synthesizer: broadly speaking, *enumerating* means that the synthesizer enumerates programs by expanding a grammar that represents the space of programs to search, and *generate-and-test* means that the synthesizer evaluates each enumerated program to test whether it fits the given examples. Figure 2.3 shows an overview of SNIPPY as a system, including details of our enumerating generate-and-test synthesizer.

As with any synthesizer, the astronomical size of the search space is the main challenge, which we begin to mitigate using the known technique of *observational equivalence* [177, 6], which unifies all programs that behave the same on inputs from the example. However, our setting adds several additional challenges: (i) Python is a dynamically typed language, so types

$$\begin{array}{lll}
E & ::= & I \mid S \mid L \mid \ldots \\
I & ::= & x \mid I\text{+}I \mid S\texttt{.find(}S\texttt{)} \mid \texttt{max(}IL\texttt{)} \mid \texttt{int(}S\texttt{)} \mid \ldots \\
S & ::= & s \mid S\text{+}S \mid S\texttt{[}I\texttt{]} \mid S\texttt{[}I\texttt{:}I\texttt{]} \mid \texttt{str(}I\texttt{)} \mid S\texttt{.join(}SL\texttt{)} \mid \ldots \\
L & ::= & SL \mid IL \\
SL & ::= & sl \mid S\texttt{.split(}S\texttt{)} \mid \texttt{[}S\texttt{ for var in }L\texttt{]} \mid \ldots \\
IL & ::= & il \mid \texttt{[}I\texttt{ for var in }L\texttt{]} \\
x & ::= & \text{vars with only int values in examples} \\
s & ::= & \text{vars with only string values in examples} \\
sl & ::= & \text{vars with only list-of-string values in examples} \\
il & ::= & \text{vars with only list-of-int values in examples}
\end{array}$$

Figure 2.4. Fragment of the expression grammar supported by SNIPPY. *E* is the root expression; *I* are integer expressions; *S* are string expressions; *SL* are list-of-strings expressions; *IL* are list-of-ints expressions.

cannot be used out of the box to constrain the search, (ii) our synthesizer generates Python list comprehensions, which are loops, a hard problem for synthesis, (iii) our synthesizer needs to generate string constants to enable the string manipulation code that Python programmers write, and (iv) our synthesizer should work with as few examples as possible, despite PBE's propensity for trivial solutions for such tasks.

Yet, despite all these challenges, we need to create a synthesizer that operates in an interactive setting, able to generate useful snippets in seconds. We discuss how we address each of these four challenges, while maintaining interactive speeds, in each of the following subsections.

2.5.1 Python as a typed language

Python is a dynamically typed language, i.e., type-checking is deferred until runtime. This means a lot of flexibility, even allowing a variable at one statement in the program to take on different types each time the statement is executed. However, our synthesizer runs statically (i.e., before runtime), at which point type information in Python is not available. Without type information for the generated expressions, any possible operation could be applied to a given expression, which makes the search space intractably large. Instead, we treat Python as a typed language with integers, strings, booleans, and lists and dictionaries that are homogeneous in type.

We do this by inferring variable types from the examples, and designing a grammar that has productions for each type, restricting the parameters of functions to certain types. To provide a sense of what this grammar looks like, Fig. 2.4 shows a fragment of the expression grammar used in SNIPPY (note it is far from the full grammar). This grammar only allows for homogeneous lists (lists of all integers or lists of all strings), and also restricts certain functions to operate only on certain types, e.g., `max` is restricted to lists of integers even though `max` can run on iterable types of all kinds. While this limits the SNIPPY synthesizer, it also greatly reduces the number of programs that must be searched, which lets SNIPPY find large and useful programs within seconds.

2.5.2 List and Dictionary Comprehensions

Our synthesizer is a *bottom-up* synthesizer, meaning that it uses production rules from the grammar to combine previously discovered expressions into larger expressions. For example, using the grammar in Fig. 2.4 and the synthesis call in Fig. 2.1(c), the enumerator starts with the terminal production rules, enumerating constants, including `' '`, and the variables, in this case the string `s`. The enumerator then applies the rule $S ::= S$`.split(`S`)` to create all expressions of this form, with S replaced by any of the current string expressions, which are the string constants and `s`, generating (among others), `' '.split(s)` and `s.split(' ')`. The process continues, iteratively building larger and larger expressions. Each expression is tested against the examples, and if an expression that satisfies all examples is found, it is returned to the user.

Unfortunately, this process breaks down for list and dictionary comprehensions that SNIPPY must support. Consider the production $SL ::=$ `[`S `for var in` L`]` for making a list of strings from another list. The nonterminal S (the "body" of the comprehension) can include the new variable `var`, so S in this production is actually derived from a different grammar than L (since L cannot access `var`). To account for this, our synthesizer uses the approach proposed in [141]: expressions are generally built using a standard bottom-up approach, except bodies of comprehensions, which are built using *nested* bottom-up enumerations. For $SL ::=$ `[`S `for var in` L`]`, the enumerator builds L bottom-up, and for each generated L it fixes L in the expression `[`S `for var in` L`]`, and then starts a nested enumeration for S.

Performing this new *nested* enumeration for comprehension bodies has three benefits: (1) the nested enumeration can include the comprehension variable `var` (2) the nested enumeration can omit the production rules for comprehensions, preventing nested comprehensions and reducing the search space, and (3) most importantly, this nested enumeration can incorporate the inputs from the examples into *observational equivalence*, as the external enumeration does, with the help of another technique called *input extension* [141], and by doing so drastically reduce the search space of the nested enumeration.

2.5.3 Discovering string literals

Bottom-up enumeration has to start with a set of constant literals like `0` and `1`. To make synthesis efficient this set must be small. SNIPPY supports the constants `-1`, `0`, `1`, and `' '`, though more constants can be enumerated with simple post-processing. E.g., the expression `1 + 1 + 1` (generated by the grammar) can be simplified via post-processing to `3`.

However, for string problems, particularly string wrangling, there is often a need for string literals that *cannot* be discovered by the grammar, even with the above post-processing. One could, in theory, ensure the grammar contained string concatenation and every ASCII character, which would let the synthesizer construct any ASCII string, but this would make the search unusably slow, and still only cover English strings.

Another approach to this problem, taken by the benchmark suite of the competition for syntax-guided solvers [9], is to adapt the synthesizer's grammar for every problem, adding only the string literals needed on a per-task basis. However, this does not suit a live system like SNIPPY where the user can ask for arbitrary problems to be solved, since we do not want to burden the user with specifying string literals each time.

SNIPPY implements a middle ground between these approaches. Its grammar is initialized with a single string literal, `' '`, but is extended as necessary. Before enumeration, the synthesizer searches the outputs of the provided examples for substrings that do not appear in any of the inputs or the grammar, and adds them to the grammar. For instance, in our motivating example, the output initials are separated by the character `'.'`, which is not part of the SNIPPY grammar. If it does not appear in any of the input variables in the example, then it will be added to the grammar. However, if the user had added a new variable `dot` with the value `'.'`, then a new constant is no longer needed, and it will not be added.

2.5.4 Variable usage

Synthesizers often require that the synthesized expression use all variables available, a property called *relevancy* [69]. However, this requirement does not make sense for SNIPPY. The example inputs include every variable in scope during the assignment statement being synthesized, including inputs to previous steps and intermediate results. Forcing the synthesizer to use all these variables can lead to unintuitive and hard to explain results, and will likely cause no result to be found.

However, removing the relevancy requirement entirely is also problematic. Let us assume the user gives the variable `out` the value `-2`. Several programs will evaluate to `-2` on the inputs from the examples, but the first one found by the synthesizer is `-1 + -1`, which is then post-processed into `-2` and returned to the user. PBE tools approach this scenario in one of two ways: by requiring the user to add another example to show `-2` is not *always* the output, or by biasing the synthesis process heavily against constants. This bias is sometimes so severe that, for instance, if `-2` is needed within a larger expression like `s[-2]` (the second to last character of the string `s`), the synthesizer would prefer a program where `-2` is generated with as few constants as possible, such as `a.find(b) + a.find(b)` when `b` is not a substring of `a`. The result, `s[a.find(b) + a.find(b)]`, is both less general—it works for the current inputs, but may not work for others—and makes little sense unless the user understands this biased model.

SnipPy does not bias its search against constants in general, but applies a reduced relevancy requirement by not returning a result program that does not use variables. For example, it will construct the expression `-2`, and use it to construct larger programs such as `s[-2]`. However, if the example output is `-2`, the synthesizer will not return `-2` as the target program, and instead continue searching for a more suitable program.

2.6 Study Methodology

To evaluate SnipPy, we conducted a within-subjects user study comparing Python development using SnipPy to developing with Projection Boxes.

We focused our study on the following research questions:

RQ1: Does SSL-PBE make a difference in speed and correctness compared to an unaided development process?

RQ2: How useful is SSL-PBE, as measured for example by the percentage of the final code that is written by SnipPy vs by the user?

RQ3: Do users report positive experiences with SSL-PBE?

RQ4: How does SSL-PBE compare to searching the internet for help?

We recruited 13 participants, 9 male, 4 female, with between 3 and 23 years of programming experience (average 8.7) for a two-hour user study. We asked potential participants to self-rate their Python experience on a scale of 1 (not familiar at all) to 5 (extremely familiar), and selected participants with experience between 2 and 4.

2.6.1 Tasks

Each participant solved 4 Python tasks from the competitive programming website codewars.com. The tasks are:

A: `abbreviate`[2]: Convert full name to lowercase initials separated by periods

[2]https://www.codewars.com/kata/554b4ac871d6813a03000035, an additional step asking for a lowercase abbreviation was added to make the task more difficult.

B: `count-duplicates`[3]: return number of characters that appear in a given list more than once

C: `max-min`[4]: compute min and max of a list

D: `palindrome`[5]: compute whether a string can be a palindrome if rotated by one ore more characters

We grouped the tasks into two sets that provided the same level of difficulty: (A,B) and (C,D). A and C were easier tasks, while B and D were harder tasks. We used two order of the tasks: (A,B);(C,D) and (C,D);(A,B).

2.6.2 Control and Test Conditions

We use two tool configuration, one control and one test. The control is called PROJECTION BOXES, which in this case will refer to the live visualization *without* SNIPPY. The test condition is SNIPPY. Since users had never seen PROJECTION BOXES before, we randomized the order of the control/test to prevent any advantage to SNIPPY users from being more experienced with PROJECTION BOXES.

Since we have two orders of the control/test, and two orders of the tasks, we have four groups:

1. SNIPPY: (A,B) ; PROJECTION BOXES: (C,D) (4 users)
2. PROJECTION BOXES: (C,D) ; SNIPPY: (A,B) (2 users)
3. SNIPPY: (C,D) ; PROJECTION BOXES: (A,B) (4 users)
4. PROJECTION BOXES: (A,B) ; SNIPPY: (C,D) (3 users)

Participants were randomly assigned into the above groups, maintaining even group sizes, divided by level of expertise. Participants were then asked to solve the first two tasks with the first tool and the second two tasks with the second tool.

[3]https://www.codewars.com/kata/54bf1c2cd5b56cc47f0007a1
[4]https://www.codewars.com/kata/554b4ac871d6813a03000035
[5]https://www.codewars.com/kata/5a8fbe73373c2e904700008c

2.6.3 Study Session

The study was conducted remotely via video conferencing. Because SNIPPY requires installing and setting up a runtime environment, the study was also conducted via remote control.

Users were first given a survey about their background as programmers. Additionally, users were asked whether they have experience with other synthesis tools, either by prior use of academic tools or using smart code completion products.

We developed instructional videos, one for PROJECTION BOXES, and one for SNIPPY. The SNIPPY video assumes PROJECTION BOXES had been introduced. Participants starting with PROJECTION BOXES were shown the PROJECTION BOXES video before using PROJECTION BOXES, then the SNIPPY video before using SNIPPY. Participants starting with SNIPPY were shown both the PROJECTION BOXES and the SNIPPY video before starting with SNIPPY, and no additional video before using PROJECTION BOXES. After the instructional video for a tool participants were given a demo task not related to the study tasks for a few minutes of guided exploration of the tool. Users were also given an opportunity to ask questions about the tool after the demo tasks.

Participants then performed the tasks. When using PROJECTION BOXES, participants were given a web browser and free internet access to search for code, whereas SNIPPY users were only given SNIPPY. Participants were instructed to use SNIPPY as much (or as little) as they wish. Tasks included suggested examples to help users check their answers. Participants determined when a task ended, either by saying they completed it or by giving up on the current task and moving to the next task. Each task was capped at 35 minutes.

After all four tasks, users were given a final survey asking them to reflect on ways SNIPPY helped them to write code.

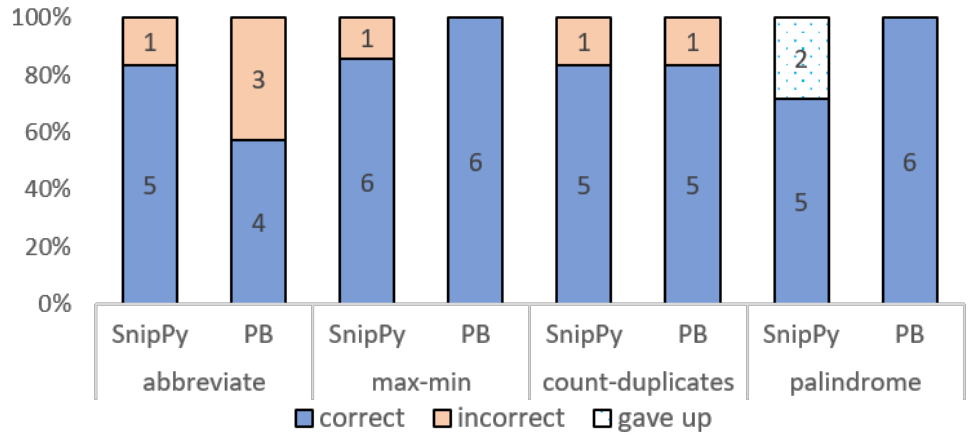


Figure 2.5. Percentage and number of correct answers for each task.

Table 2.1. Changes in session times in SNIPPY compared to PROJECTION BOXES. Negative percentage indicates a speedup.

		Easy		Hard	
		`abbreviate`	`max-min`	`count-duplicates`	`palindrome`
All	avg	22%	33%	-7%	2%
	med	21%	139%	-18%	-21%
Correct	avg	21%	13%	-25%	-1%
	med	28%	80%	-36%	-21%

2.7 Results

2.7.1 Session times and correctness (RQ1)

Fig. 2.5 shows the number and percentage of correct answers to each task, determined via 10 unit tests for each task that were run after the session ended.. Our study is too small to show statistical significance, but we examine the tasks with the most notable differences: `abbreviate` and `palindrome`.

In `abbreviate`, participants who did not have SNIPPY made more mistakes. Most of the mistakes in `abbreviate` had to do with using an incorrect separator between initials. Users who used SNIPPY to synthesize the code that combines the first letters of the names did not make this mistake, as they used the given expected output to generate the correct code. Also, of the two participants who gave up on `palindrome` (P1, and P12), P12 did not use the synthesize

Table 2.2. Synthesis calls in relation to the final solution

		abbreviate	max-min	count-duplicates	palindrome
$\frac{\text{Useful calls}}{\text{Total calls}}$	avg	61%	36%	36%	27%
	med	58%	25%	20%	20%
$\frac{\text{Synthesized}}{\text{All code}}$	avg	47%	66%	28%	18%
	med	47%	65%	23%	15%

function feature of SNIPPY for the entire 20 minute session, essentially reducing the session to a PROJECTION BOXES session.

Tab. 2.1 shows the change in session times from PROJECTION BOXES to SNIPPY, with PROJECTION BOXES as a baseline. A negative percentage indicate a reduction in session time (speedup), whereas a positive percentage indicates an increase in session time (slowdown). The numbers are provided both for all sessions, and for all sessions where the participants found correct solutions.

While our study is not large enough to provide statistically significant results, broadly speaking, our preliminary numbers suggest a possible pattern based on how hard the task is. Indeed, recall that `abbreviate` and `max-min` were easier tasks, while `count-duplicates` and `palindrome` were harder tasks. In the two easier tasks, SNIPPY appears to make the sessions longer, whereas for the two harder tasks, SNIPPY appears to make the sessions shorter.

There are two factors that could explain this. First, for easier tasks, writing the code directly can be faster than using a synthesizer, especially if using the synthesizer requires multiple round trips (e.g., if the first example is insufficient, and a second example is needed).

Our results echo those of previous studies such as Galenson et al. [57], where using synthesis in a freeform manner more than doubled the time to completion of the task. However, we are encouraged by the fact that our slowdown is not as severe, indicating the live programming aspect of SSL-PBE helps mitigate some of the overhead of using the tool.

2.7.2 Usefulness of Synthesis (RQ2)

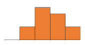
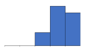
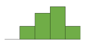
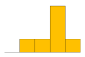
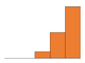
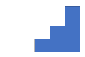
We measured how many synthesis calls were useful to the programmer. We counted as useful any synthesis call where a non-trivial part of the synthesized code was used in the participant's final program. As not useful we counted all other calls, including calls where synthesis timed out. The results are in the top part of Tab. 2.2.

In general, we see that a sizeable number of synthesis calls are not useful. Still, for every task but `palindrome` at least one participant had 100% useful synthesis calls. P8 in `count-duplicates` and P12 in `palindrome` had no useful calls in the course of solving their task.

Additionally, we measured how much of each user's final program originated from the synthesizer. To do this, we computed the proportion of tokens (using Python's own code tokenizer) in the user's answer that came from synthesis. We did not count things not generated by SNIPPY, such as the assignment into a variable or the return statement of the function, and in the case the user renamed a variable in the synthesized code, the variable name was counted as user code, while the rest of the snippet was counted as synthesizer code. In short, we measured the manual effort that was performed by the user and how much was delegated to the synthesizer. The average and median results are in the bottom part of Tab. 2.2.

Because SNIPPY does not generate things like the return statement or assignments, and because `palindrome` required a loop to be manually written, 100% synthesized was not a possible result. Broadly speaking, programmers tackling the harder tasks (`count-duplicates` and `palindrome`) wrote more of the program manually. These tasks are harder to break up into synthesis-ready chunks, and in some approaches to the task, the synthesizer will no longer help. All tasks except `palindrome` *could* be solved almost entirely by synthesis, and the largest portion synthesized by one user was 83% in `count-duplicates` (P3, 39 of 47 tokens). The way a problem is deconstructed for synthesis is crucial to how much of it can be synthesized. Users whose breakdown of the task meshed with SNIPPY could synthesize every step and write

Table 2.3. Survey Results. All questions are on a likert scale where 1 is "Disagree" and 5 is "Agree".

	Average	Median	Dist.
SNIPPY helped me write my code	3.46	3	
SNIPPY was easy to use	4.23	4	
I would use SNIPPY again	3.54	4	
SNIPPY would be useful beyond today's tasks	3.69	4	
I would like to have PROJECTION BOXES	4.54	5	
I would like to have SNIPPY available	4.38	5	

almost no code, whereas users who did not come up with such a breakdown were still able to synthesize code, but to a lesser extent. Overall, we see that although a lot of synthesis calls were not successful, calls that were successful provided users with substantial parts of the solution.

SNIPPY and data-dependent loops

A very frequent cause of failed synthesis calls was an attempt to synthesize statements inside loops that cause a data dependency between the iterations, or a loop where a variable is written to in one iteration, then used in the next, a simple example of which is `sum = sum + i`. Dependent loops are a known hard problem in program synthesis [141], and are notoriously hard to specify correctly even under the best conditions. Attempting to synthesize these was a gap in the participants' understanding of the synthesizer limitations (even for participants who were previously familiar with synthesis tools). We discuss the implications of this gap in the next section.

2.7.3 User survey (RQ3)

Tab. 2.3 shows the results of our survey, including the average, median and the distribution of scores. In the "Discussion" section below we will discuss in more detail the factors that affect the utility of SNIPPY, and explain these results. For now, we do note that, even though the scores on utility are lower than others, because SNIPPY can be invoked as needed, users still overall said they would like to have SNIPPY available.

2.7.4 Comparison to Searching the Internet (RQ4)

One of the questions in our post-study questionnaire asked participants to compare SNIPPY to searching the internet. Overall, 23.1% of participants said that they preferred SNIPPY to the internet, 15.4% said they preferred the internet, and the remainder said that it depends and explained the trade-offs.

P1, P10, and P12 stated that SNIPPY can work well even if one does not have a clear picture of what they should search for online. P1 also said that SNIPPY solutions are more concise.

One recurrent theme we observed is that searching the internet and SNIPPY supplement each other, each having different strengths. (In fact P7 said that they would first try SNIPPY and if that didn't work they would search the internet.) For the kinds of code snippets that SNIPPY can generate, SNIPPY is better, for several reasons that were explicitly mentioned by our participants. First, SNIPPY can find a solution quickly without imposing the cognitive burden of switching to another window or tool. Second, SNIPPY can find compact solutions. Third SNIPPY correctly connects the generated snippet to the surrounding code – in contrast solutions from the internet often need to be adapted and correctly glued into the surrounding context, a non-trivial and error prone task. Finally SNIPPY can work well even when the programmer does not have a clear picture of what to search for on the internet.

On the other hand, however, SNIPPY (as with any synthesis tool) has limitations in what it can do, and this affects its utility compared to searching the internet.

2.8 Discussion

2.8.1 Usage of Small-Step Live Programming by Example

Through our study, we identify three predominant ways in which SNIPPY helped programmers.

First, some participants used SNIPPY in precisely the way we anticipated: decomposing the problems into smaller steps, then editing the live data to make SNIPPY generate code snippets for those smaller steps. In these cases, SNIPPY does not help *algorithmically*, but instead provides help with individual steps of a larger algorithm. The most successful uses of SNIPPY were ones where the programmer came up with the high-level strategy, and SNIPPY helped with the individual steps.

Second, some participants used SNIPPY "on the side": they would stop coding the main task they were working on, and start writing code separately to get SNIPPY to generate a useful snippet. For example, P1 used this approach to generate code for rotating a string by a constant number—3 characters. Once the code for rotating a string by 3 was generated, P1 took the snippet, generalized it to an arbitrary rotation by `k` and placed it inside a loop. This interruption in the flow of programming leads to a less fluid process, but still uses SNIPPY effectively.

Third, some participant used SNIPPY to recall details about Python syntax or Python libraries they had forgotten. In this situation, the programmer might know how to do something, but forgot (or possibly is not fully familiar with) the details of expressing it in Python. Examples of such easily forgotten details, especially for those with less Python experience (but even for programmers with a lot of Python experience) include: the order of parameters to certain methods, like `split`; the exact syntax of dictionary comprehension; the exact syntax of list/dictionary comprehension with an embedded filter; the name of library functions, e.g., for converting characters to lower case, for returning the keys *and* values of a dictionary, or for returning the elements *and* indices of a list.

2.8.2 Understanding Synthesized Code

When a synthesizer generates code, there is a question of how well the programmer understands the code. In our study, programmers checked that the code appeared reasonable but did not try to understand the details. In some cases, participants remarked on the synthesized code being simpler than they would have written. In other cases, participants explicitly said that the code worked, but they did not fully understand it.

One may be concerned about correctness when programmers use code that they do not fully understand, but in our study we observed that this did not drive programmers to an incorrect solution. We also observed users sometimes take code snippets they do not fully understand from the internet in our control setting, and are not the first to document this [22].

However, we observed a much more interesting problem when programmers do not understand the synthesized code: it leads to the mindset that the synthesizer is all-or-nothing: either the synthesizer eventually generates code that works, or if not, then the programmer just gives up on the synthesizer altogether. Unfortunately, this can prevent the programmer from using an almost-correct solution generated by the synthesizer.

This happened to P2 who used SNIPPY in `palindrome` to generate an almost-correct solution. Given the setup that the programmer used, the synthesized code only worked for lists of size 4. Had the programmer generalized 4 to an expression for the list's length, the problem would have been solved. Instead they tried unsuccessfully to generalize the examples and re-synthesize, and eventually gave up on the problem.

More generally, this leads us to the following takeaway:

Because programmers do not try to understand the code generated by the synthesizer, they unnecessarily shy away from trying to use partial results from the synthesizer.

This in turn points to a possible direction for future research, namely on understandability and usability of partial results in synthesizer-generated code (something that has already started being explored, for example in Wrex [41] and Bester [142]).

2.8.3 Mental Model of the Synthesizer

We have noticed that the mental model that the programmer has of the synthesizer is very important. We start by framing our discussion in terms of the well-known gulfs of evaluation and execution. The gulf of evaluation captures how well a user can understand the internal state of the system. The gulf of execution captures how well a user can discover how to make the system take steps toward an ulterior goal. In the setting of programming, the gulf of evaluation relates to understanding the program state and its result; the gulf of execution relates to understanding what kinds of statements should be written next to finish a task. E.g., at a command line prompt, showing the current directory and the computer name eases the gulf of evaluation (exposing internal state); making commands at the prompt discoverable via auto-complete or command-line searches may ease the gulf of execution (making it easier to choose the next step toward a goal).

PROJECTION BOXES help with the gulf of evaluation, since they make the internal state of the program visible at all times. However, they do not help explicitly with the gulf of execution, since they do not help directly with writing the code.

SNIPPY provides this missing aspect of PROJECTION BOXES, easing the gulf of execution with explicit help discovering the next statement toward a broader end goal. However, this over-simplification misses an important subtlety. While SNIPPY does ease the gulf of execution in some ways, it introduces a different kind of burden that also relates to the gulf of execution: the programmer must now pick between SNIPPY and one of three other approaches: (1) writing the code by hand, (2) searching the internet, or (3) manually decomposing the problem into smaller pieces to try with SNIPPY.

So, in essence, we have shifted the gulf of execution from one kind of gulf to another: from figuring out what statement to write next, to figuring out if SNIPPY should be used for the next statement. This new gulf of execution is particularly interesting because for programmers to make the choice between SNIPPY and other approaches, we have observed that they must have an *accurate* mental model of the synthesizer's abilities. If a programmer broadly understands (through trial and error) what kinds of tasks the synthesizer can do, they will know when to invoke the synthesizer and when to try something else. However, if the programmer has a poor mental model of the synthesizer's ability (e.g., one that *overestimates* the synthesizer's ability), then the programmer might waste time and energy trying to get the synthesizer to do something that it simply cannot. This leads to frustration, making it less likely that the synthesizer will be used the next time around. Furthermore, if the programmer *underestimates* the synthesizer's ability, they will under-utilize the tool.

> *We introduce the term* **user-synthesizer gap** *to refer to this gap between the user's mental model of the synthesizer's abilities and the actual abilities of the synthesizer.*

We are not the first to notice this kind of effect. Lau [98] explored the related topic of a user's *trust* of the synthesizer, concluding that the adoption of Programming by Demonstration tools is held back by tool behaviors that undermine that trust. Gero et al. [58] explored mental models of AI agents in an interactive game, strengthening the conclusion that mistrusting the system is detrimental to success of the user, but also finding that: (1) users with a generally good understanding of AI systems developed a better mental model of the AI agent and (2) people tended to overestimate the AI's abilities.

The way the *user-synthesizer gap* manifested itself in our study shows that the overestimation of the AI system's capabilities documented by Gero et al. also occurs for synthesizers, but that it may involve *underestimating* the synthesizer's ability instead. Because state-of-the-art general-purpose synthesizers still cannot generate *all* the necessary code in a real setting, the only way a synthesizer can help a programmer is on sub-problems to a larger task. In this situation, the *user-synthesizer gap* will inevitably come into play. This is less pronounced in domain-specific tools, as the limits of the domain act as an accurate mental and actual model for the synthesizer's limits.

We observed three properties about this gap. First, it is a much bigger problem if the user over-estimates the synthesizer's ability than underestimates it. Second, the larger the gap is, the more difficult it becomes for the programmer to make choices about how to incorporate the synthesizer into programming tasks. Third, this gap is self-correcting in some ways, in that if the gap is large, programmers eventually understand this, and adjust their mental model to reduce the gap. Consequently, as the programmer learns more about the synthesizer through trial and error, the gap can decrease over time, but this is a non-trivial learning curve that takes time, and can be a significant impediment to the adoption of synthesizers.

All the above observations lead us to believe that reducing the *user-synthesizer gap* represents an impactful future research arc that has the potential to further unlock the potential of state-of-the-art synthesis techniques.

2.9 Limitations

While the results of the study are promising, our study has certain limitations that remain to be addressed in future work.

Our study compares SSL-PBE with live programming augmented with searching the internet in a browser. Further studies would be needed to compare SSL-PBE to other interaction models, such as web searches or knowledge bases embedded in the IDE [107, 46], big-step synthesis tools in and out of the IDE [81], and smart code completion [72].

There are also several threats to the validity of our results. Our survey was conducted in the presence of one of the authors, which could lead to a social desirability bias. Additionally, the phrasing of questions was not neutral (e.g. "SNIPPY helped me write my code" instead of "How helpful was SNIPPY in writing your code").

There may also be a bias in our findings on users' understanding of the synthesized code. Our tasks each included one or more examples participants could input into the live programming environment, which could limit users' view of the code to those inputs and discourage them from examining the synthesized code further.

Finally, the small sample size and short length of tasks could be a threat to the internal validity of the study. Individual differences in coding speed could affect the conclusions we drew from the length of the programming sessions.

2.10 Conclusion

We introduced a new paradigm called *Small-Step Live Programming by Example* and discussed its implementation in SNIPPY. Through a within-subjects study we demonstrated that this paradigm is easy to use, and is most effective in non-trivial tasks. We also found that almost all participants preferred SNIPPY over searching the internet in some cases. Furthermore, our study showed that most users did not attempt to understand the code deeply, which resulted in an all-or-nothing approach to using SNIPPY's output. Finally, we identified the "*user-synthesizer gap*", which describes the gap between the user's mental model of the synthesizer's capabilities and its actual capabilities. We believe that reducing this gap represents an important direction for future research.

2.11 Acknowledgements

Chapter 2, in full, is a reprint of Small-Step Live Programming by Example, as it appears in the Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20). It was done in collaboration with Allen Ordookhanians, Hila Peleg, Sorin Lerner and Nadia Polikarpova. The dissertation author was a primary investigator and author of this paper.

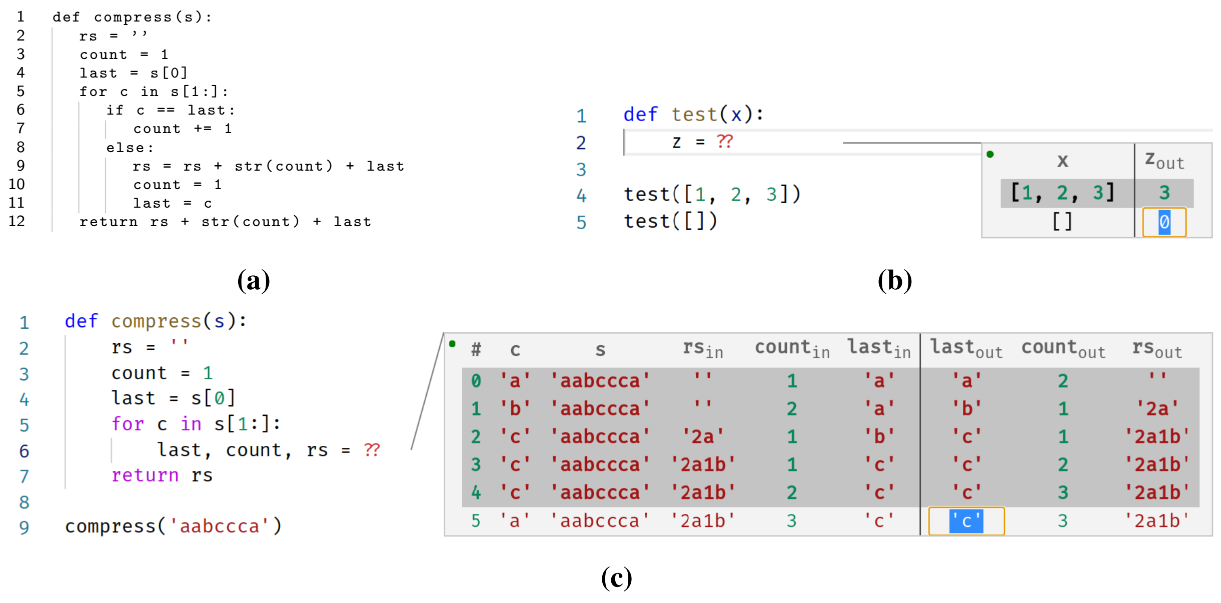
Chapter 3

LooPy: Bottom-up Enumerative Synthesis with Control Structures

3.1 Introduction

As software development environments continue to evolve, one feature that has long captured the community's imagination is a "programmer's assistant", watching over the programmer's shoulder and helpfully suggesting code snippets to solve small tasks that continuously arise during development. The assistant would allow the programmer to focus on the core algorithm instead of getting bogged down in low-level details or interrupting their flow to search for code online. In recent years, this dream seems to be within reach thanks to algorithmic advances in *program synthesis* and specifically *programming by example* (PBE) [185, 47, 117, 14, 161, 16, 15, 165]; despite these advances, however, the programmer's assistant remains elusive. Where do existing PBE synthesizers fall short?

Big-step synthesis. Consider a Python programmer who wants to solve the String Compression task from the popular book *Cracking the Coding Interview* [123, p. 91]: *Implement basic string compression using the counts of repeated characters. For example, the string `"aabccca"` would become `"2a1b3c1a"`.* Traditional PBE synthesizers adopt a *big-step* interaction model, where the programmer specifies inputs and outputs at the function level, and the synthesizer is expected to generate the entire function body in one shot. In our example, the programmer might provide the input-output example `"aabccca"` $\to$ `"2a1b3c1a"`, and expect the synthesizer



```
1   def compress(s):
2     rs = ''
3     count = 1
4     last = s[0]
5     for c in s[1:]:
6       if c == last:
7         count += 1
8       else:
9         rs = rs + str(count) + last
10        count = 1
11        last = c
12    return rs + str(count) + last
```

(a)

```
1  def test(x):
2      z = ??
3
4  test([1, 2, 3])
5  test([])
```

x	z_{out}
[1, 2, 3]	3
[]	0

(b)

```
1  def compress(s):
2      rs = ''
3      count = 1
4      last = s[0]
5      for c in s[1:]:
6          last, count, rs = ??
7      return rs
8
9  compress('aabccca')
```

#	c	s	rs_{in}	$count_{in}$	$last_{in}$	$last_{out}$	$count_{out}$	rs_{out}
0	'a'	'aabccca'	''	1	'a'	'a'	2	''
1	'b'	'aabccca'	''	2	'a'	'b'	1	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'c'	1	'2a1b'
3	'c'	'aabccca'	'2a1b'	1	'c'	'c'	2	'2a1b'
4	'c'	'aabccca'	'2a1b'	2	'c'	'c'	3	'2a1b'
5	'a'	'aabccca'	'2a1b'	3	'c'	'c'	3	'2a1b'

(c)

Figure 3.1. (a) Motivating example: Python solution for String Compression. This program loops over the input string `s`, updating the result `rs` and two auxiliary variables: `last`, the previous character from `s`, and `count`, the length of the current run. (b) Prior work: Small-Step Live PBE. The user enters desired values for `z` into a projection box, and the synthesizer replaces `??` with `len(x)`. (c) Our work: block-level synthesis with LoopY. The user enters values for `last`, `count`, and `rs`, and the synthesizer replaces line 6 with the entire loop body.

to generate the body of the function `compress` in Fig. 3.1a. Unfortunately, this function contains several features that make it challenging for program synthesis techniques to discover: it is relatively long and contains both library function calls and a loop. Although state-of-the-art PBE synthesizers such as FrAngel [161] are capable in principle of generating programs of this complexity,[1] they require minutes, not seconds, to do so, and the result can be unpredictable and sensitive to the provided examples, making big-step synthesizers unsuitable for the interactive setting of a programmer's assistant.

Small-step synthesis. To enable program synthesis in interactive settings, a different line of work [57, 52] has developed a *small-step* interaction model, where the synthesizer—typically integrated into the IDE—is used to generate just the next line of code, and the programmer is

[1]We attempted to solve the String Compression task with FrAngel by providing up to nine input-output examples of varying complexity, but it failed to find the right solution within a 30 minute timeout.

expected to provide a local specification for that line. In particular, the interaction model we developed in [52], dubbed *Small-Step Live PBE*, uses a live programming environment—an enviroment where the program state is continuously displayed—named *Projection Boxes* [105]. With Small-Step Live PBE, the programmer may edit the live state to specify desired values for a single output variable, prompting the synthesizer to generate an assignment to that variable; for example in Fig. 3.1b, when the programmer enters desired values for `z` into the projection box for the two rows representing live values from the different invocations of `test`, the synthesizer replaces the prompt `??` with a generated expression `len(x)`.

An important advantage of this interaction model is that the programmer needs to specify only the *after-state* for the synthesis problem, while the *before-state* is supplied by the live programming environment from the live state before the update, seen on the left of the projection box. And because the code for each individual assignment is smaller and simpler than a full big-step solution, the synthesizer can produce useful results in seconds, making it suitable for interactive use.

Unfortunately, Small-Step Live PBE would not be very helpful when solving the String Compression task, because this task requires control structures (loops and conditionals). Intuitively, it is hard to specify code inside a loop one assignment at a time because of the *dependent before-state* problem [141]: the before-state of a given loop iteration depends on the code executed in the previous loop iterations. In our example, suppose the programmer is in the middle of implementing `compress` from Fig. 3.1a and is yet to write the entire `else` branch; if they now try to synthesize the assignment to `rs` in line 9 by providing the value of `rs` for the first few loop iterations, this will fail because the right-hand side of this assignment uses the variables `rs`, `count`, and `last`, which all should be updated in the loop; hence the synthesizer does not have access to the correct before-state for these variables for later loop iterations.

Our approach: Block-Level Live PBE. To overcome this limitation and enable interactive synthesis in the presence of control structures, we propose a new *block-level* interaction model, which we call *Block-Level Live PBE*, and implement this model in a synthesizer called LOOPY.[2] LOOPY builds on Small-Step Live PBE, but allows the programmer to specify the after-state for *multiple* output variables at a time, prompting the synthesizer to generate a *code block*, *i.e.*, a sequence of assignments, possibly including conditionals. In our running example, the programmer can use LOOPY to generate the entire loop body, by *simultaneously* specifying the values for `last`, `count`, and `rs` for the first five loop iterations, as shown in Fig. 3.1c. Given this input, LOOPY generates the code on lines 6–11 of Fig. 3.1a in under two seconds. A video showing LOOPY is found at https://youtu.be/EIWtF4BJpmo.

The key technical insight that makes Block-Level Live PBE effective is *live execution*, a concept inspired by the live evaluation of [117]. In live execution, the programmer performs the natural act of providing variable values for loop iterations *in order*. In doing so, the programmer essentially serves as an interactive *oracle* to execute missing statements. As such, live execution can accurately propagate the before-state through a loop, even when the loop is not yet complete, which solves the dependent before-state problem. Indeed, given the specification in Fig. 3.1c, LOOPY can use the programmer-provided after-state at iteration `0` as the before-state for iteration `1`,[3] and similarly for the next three iterations.

Although synthesizing the entire loop body at once is often convenient, we deliberately designed LOOPY to be flexible with respect to the granularity of the block. In particular, it also supports synthesizing loop bodies one assignment at a time, as well as mixing hand-written and synthesized code (as long as the output variables of each synthesis problem only depend on variables that are already correctly updated or are part of the same synthesis problem). For example, the programmer might manually write the code that updates `last` and `count`, and then use LOOPY to synthesize the assignment to `rs` on line 9, and LOOPY will correctly compute and display the before-state of future loop iterations by taking into account both the specified after-state for `rs` and the hand-written code for the other variables.

[2]The code for LOOPY is found at https://github.com/KasraF/LooPy, and as a VM image at https://doi.org/10.5281/zenodo.5459013.

[3]Iteration counts are displayed in the "#" column of the projection box.

Efficient synthesis of code blocks. The core technical challenge in building a block-level synthesizer like LOOPY is to make synthesis scale at interactive speeds to larger code snippets, such as the loop body in Fig. 3.1a (lines 6-11). To avoid blindly enumerating all sequences of assignments, we leverage our block-level interaction model: given a complete before- and after-states for a synthesis problem, LOOPY can enumerate single assignment subprograms considering all intermediate states that the program might go through. We introduce a data structure called the *Intermediate State Graph* (ISG), which compactly represents all paths from the before-state to the after-state through those intermediate states.

Evaluation. We empirically evaluate the performance of our new synthesizer LOOPY, and show that it can handle a wide range of synthesis tasks at interactive speeds. Through a small-scale qualitative user study with five participants, we also evaluate the feasibility of providing block-level specifications. In our study, participants used LOOPY to solve two Python programming tasks that involved loops. Our study shows that generally programmers are able to provide block-level specifications.

Contributions. In summary, this paper makes two main contributions:

1. A new interaction model called *Block-Level Live PBE*, whose key technical insight is *live execution*, an approach that uses programmer input as an oracle to compute the before-state of future loop iterations, even when the loop body is incomplete. Our user study demonstrates the feasibility of this interaction model.

2. A *block-level program synthesis* algorithm using *Intermediate State Graph*s. Our empirical evaluation shows that this algorithm can synthesize a variety of code blocks using small specifications and in interactive times (seconds).

3.2 Motivating example

In this section we illustrate the interaction model and the inner workings of LOOPY using the String Compression task from in Fig. 3.1 as the running example. We assume that our user is an experienced programmer, but does not use Python very often; hence they have a high-level idea of the algorithm they want to implement, but they would like to use program synthesis to figure out the details at every step.

Prior work: Small-Step Live PBE. One paradigm for integrating synthesis into the programmer's workflow is our recent work on Small-Step Live PBE [52], an approach that combines live programming in the Projection Boxes environment [105] with programming by example. Small-Step Live PBE allows the user to edit the live state of the program after a missing assignment statement, prompting the synthesizer to generate a statement that modifies the state accordingly. Fig. 3.1b shows a simple example: the user wants to compute the length of the list `x`, but they have forgotten the name of the corresponding Python function. To invoke the synthesizer, they introduce a *hole*, `z = ??`, which spawns a projection box where the output variable `z` can be edited. The user might provide the value for `z` in the first line only; this gives rise to a synthesis task that asks to transform the *before-state* $\sigma_{start} = \{\texttt{x} \mapsto \texttt{[1,2,3]}, \texttt{z} \mapsto \bot\}$ into the *after-state* $\sigma_{end} = \{\texttt{x} \mapsto \texttt{[1,2,3]}, \texttt{z} \mapsto 3\}$. If the user specifies `z` in both lines, the synthesis task becomes a set of examples (before-after pairs) $\sigma^0_{start} \to \sigma^0_{end}, \sigma^1_{start} \to \sigma^1_{end}$. Given this specification, the synthesizer generates the assignment `z = len(x)`, which replaces the hole.

The Small-Step Live PBE interaction model has two important properties. First, the user only needs to provide the after-state σ_{end} for each example; the before-state σ_{start} is computed by the live programming environment simply by *executing* the program up until the point of the hole. This saves user effort, since they need not manually construct relevant before-states from the middle of an execution. Second, the user's expectation of the synthesizer is that once the hole is replaced by the synthesis result, the program execution will indeed pass through *the exact after-states* they had specified.

Challenge: loops. These properties become non-trivial to realize in the presence of loops. Going back to the `compress` function in Fig. 3.1a, if the user wants to invoke Small-Step Live PBE inside the loop (say, to help with the assignment on line 9) the projection box cannot display an accurate before-state for any loop iteration beyond iteration `1`, because that would require executing the loop body, which has not yet been completed. While in principle it is possible to ask the user to provide after-states for *arbitrary* before-states, this violates the user's mental model of Live PBE: that they are specifying states that are a part of a single program execution, and the synthesized programs will actually pass through those states.

Our solution: Block-Level Live PBE. To extend the Live PBE paradigm to work in the presence of loops, we propose a new interaction model we dub *Block-Level Live PBE* and implement this model in a synthesizer called LooPy. In our `compress` function in Fig. 3.1a, LooPy can synthesize the entire block of code inside the for loop, lines 6–11. In the rest of the section we explain how LooPy synthesizes this code, gradually building up to it through a series of smaller-scale synthesis problems for different fragments of the `compress` function. We start with explaining how LooPy synthesizes a single assignment, then multiple assignments, and finally the whole conditional.

3.2.1 Handling Loops with Live Execution

To start, we explain how our approach works for a single assignment. Consider for example the setting where the user has already figured out how to update the auxiliary variables `count` and `last`, and only needs help with appending to `rs`. In other words, the user starts with a *sketch* shown on the right, where the hole `rs = ??` on line 9 indicates the statement they would like to synthesize. (This program is a prefix of the full solution in Fig. 3.1a; we assume that the user will add the `return` statement later by hand).

```
1  def compress(s):
2      rs = ''
3      count = 1
4      last = s[0]
5      for c in s[1:]:
6          if c == last:
7              count += 1
8          else:
9              rs = ??
10             count = 1
11             last = c
```

Live execution. To enable Live PBE in the presence of loops, LooPy performs *live execution* (inspired by the live evaluation of [117]), where the programmer serves as an *oracle* to execute missing statements and accurately propagate the before-state through the sketch.

In our example, when the user first invokes LOOPY, they see the projection box in Fig. 3.2a and are prompted to enter the output value for `rs` in iteration `1`.[4] Because the execution until that point has not encountered any holes, the projection box displays an accurate before-state for this iteration[5]:

$$\sigma^0_{start} = \{\texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{''}, \texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'}\}$$

Once the user enters the desired value `'2a'` for `rs`, LOOPY uses it as an oracle to "jump over" the missing assignment and compute the state after line 9 as (with the modified part highlighted):

$$\sigma^0_{end} = \{\texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{'2a'}, \texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'}\}$$

Starting from this state, LOOPY executes the rest of the loop body, to compute the accurate before-state for the next time the execution encounters the hole (in iteration `2`):

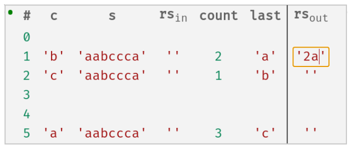
$$\sigma^1_{start} = \{\texttt{c} \mapsto \texttt{'c'}, \texttt{rs} \mapsto \texttt{'2a'}, \texttt{count} \mapsto 1, \texttt{last} \mapsto \texttt{'b'}\}$$

At this point the user is prompted to enter the after-state for iteration `2`, as shown in Fig. 3.2b, which again can be used as an oracle to continue live execution and further update the projection box (see Fig. 3.2c). After any number of iterations, the programmer can decide to stop and invoke the synthesizer with the specifications provided so far.

Thanks to live execution, the two desirable properties of Live PBE are preserved inside the loop: 1. the before-states like σ^0_{start}, σ^1_{start} are supplied by the environment, and 2. once synthesis is complete, program execution passes through the states specified by the user. From the UI standpoint, live execution is supported by enforcing that the user specify after-states for

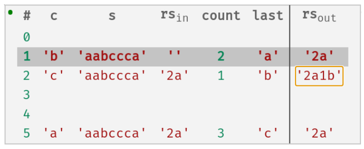
[4]Iteration `0` is missing from this box, since it takes the other branch of the conditional and hence does not execute the hole.

[5]In this section we omit `s` from the states for brevity, since it does not change.



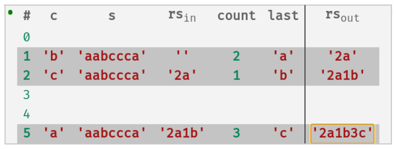
#	c	s	rs$_{in}$	count	last	rs$_{out}$
0						
1	'b'	'aabccca'	''	2	'a'	'2a'
2	'c'	'aabccca'	''	1	'b'	''
3						
4						
5	'a'	'aabccca'	''	3	'c'	''

(a) Before the first iteration



#	c	s	rs$_{in}$	count	last	rs$_{out}$
0						
1	'b'	'aabccca'	''	2	'a'	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'2a1b'
3						
4						
5	'a'	'aabccca'	'2a'	3	'c'	'2a'

(b) After the first iteration



#	c	s	rs$_{in}$	count	last	rs$_{out}$
0						
1	'b'	'aabccca'	''	2	'a'	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'2a1b'
3						
4						
5	'a'	'aabccca'	'2a1b'	3	'c'	'2a1b3c'

(c) The complete specification

Figure 3.2. Specifying after-states for the hole `rs = ??` in LOOPY. Note that the projection box only displays those iterations that execute the hole (`1`, `3`, and `5`).

each evaluation of the hole *in order*; for example, in Fig. 3.2b the user cannot skip iteration `2` and proceed to enter the value for iteration `5`, because the before-state in the latter iteration is not yet accurate.

Synthesis. Live execution reduces the sketch and the user input from the projection box into a local synthesis problem, defined simply as a set of before-after pairs. For example, user input from Fig. 3.2c generates the following pairs:

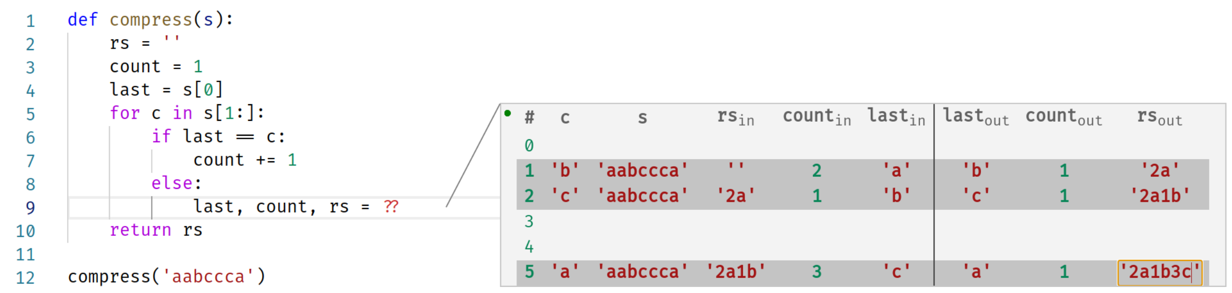
$$\sigma^0_{start} = \{\ \texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{''}, \quad \sigma^0_{end} = \{\ \texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{'2a'},$$
$$\texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'}\} \quad \texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'}\}$$

$$\sigma^1_{start} = \{\ \texttt{c} \mapsto \texttt{'c'}, \texttt{rs} \mapsto \texttt{'2a'}, \quad \sigma^1_{end} = \{\ \texttt{c} \mapsto \texttt{'c'}, \texttt{rs} \mapsto \texttt{'2a1b'},$$
$$\texttt{count} \mapsto 1, \texttt{last} \mapsto \texttt{'b'}\} \quad \texttt{count} \mapsto 1, \texttt{last} \mapsto \texttt{'b'}\}$$

$$\sigma^2_{start} = \{\ \texttt{c} \mapsto \texttt{'a'}, \texttt{rs} \mapsto \texttt{'2a1b'}, \quad \sigma^2_{end} = \{\ \texttt{c} \mapsto \texttt{'a'}, \texttt{rs} \mapsto \texttt{'2a1b3c'},$$
$$\texttt{count} \mapsto 3, \texttt{last} \mapsto \texttt{'c'}\} \quad \texttt{count} \mapsto 3, \texttt{last} \mapsto \texttt{'c'}\}$$

Given this specification, the synthesizer takes less than a second to generate the expression `rs + str(count) + last` for the right-hand side of the hole `rs = ??`, and LOOPY replaces the statement with a pretty-printed version: `rs += str(count) + last`.

To solve the local synthesis problem, LOOPY uses a popular synthesis technique called *bottom-up enumeration with Observational Equivalence reduction* [177, 6]. In this technique, an *expression enumerator* gradually builds more and more complex expressions by composing



```python
def compress(s):
    rs = ''
    count = 1
    last = s[0]
    for c in s[1:]:
        if last == c:
            count += 1
        else:
            last, count, rs = ??
    return rs

compress('aabccca')
```

#	c	s	rs_{in}	$count_{in}$	$last_{in}$	$last_{out}$	$count_{out}$	rs_{out}
0								
1	'b'	'aabccca'	''	2	'a'	'b'	1	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'c'	1	'2a1b'
3								
4								
5	'a'	'aabccca'	'2a1b'	3	'c'	'a'	1	'2a1b3c'

Figure 3.3. Specifying after-states for a hole *with multiple variables* in LOOPY.

previously enumerated simpler expressions; all expressions are evaluated on the set of before-states σ^i_{start}, and expressions with the same output are pruned.

3.2.2 Synthesizing assignment sequences with Intermediate State Graphs

Next we explain how our approach works for blocks of statements. In our running example, the order of assignments in the `else` branch of `compress` can be tricky to get right, so it would be nice to delegate the entire `else` branch to the synthesizer. In LOOPY the user can achieve this by writing a hole with multiple output variables—`last`, `count`, and `rs`—as shown in Fig. 3.3.

This spawns a projection box that prompts the user to enter values for *all three output variables*, at each loop iteration. Given the user input in Fig. 3.3, LOOPY again takes less than a second to generate the sequence of assignments shown on the right.

```python
rs += str(count) + last
count = 1
last = c
```

While live execution extends straightforwardly to multi-variable holes, the challenge is to extend the synthesis back-end to synthesize the correct assignment order while maintaining interactive speeds. Fortunately, because in our setting the entire before- and after-states are given, we can decompose the synthesis of a sequence of assignments into sub-problems for individual assignments.

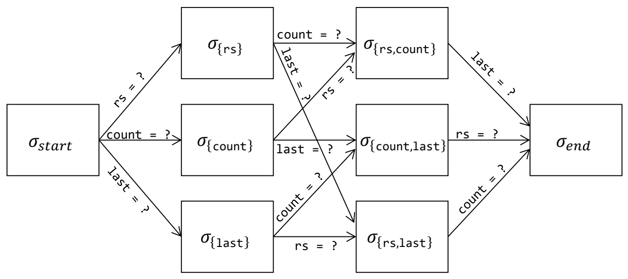


Figure 3.4. Intermediate State Graph for synthesizing `rs`, `last`, and `count`.

Specifically, consider the before-after pair for iteration `1` in Fig. 3.3:

$$\sigma_{start} = \{\texttt{c} \mapsto \texttt{'b'}, \texttt{last} \mapsto \texttt{'a'}, \texttt{count} \mapsto 2, \texttt{rs} \mapsto \texttt{''}\}$$

$$\sigma_{end} = \{\texttt{c} \mapsto \texttt{'b'}, \texttt{last} \mapsto \texttt{'b'}, \texttt{count} \mapsto 1, \texttt{rs} \mapsto \texttt{'2a'}\}$$

A sequence of three assignments that transforms σ_{start} into σ_{end} must pass through two intermediate states. Let us first assume that the order of assignments is given (first `rs`, then `count`, then `last`), and the synthesizer only needs to generate their right-hand sides. In this case, the two intermediate states are fixed: the first one is the state where only `rs` is updated and the rest of the variables have the same values as in σ_{start} (we denote it $\sigma_{\{\texttt{rs}\}}$); similarly, the second state is the one where only `rs` and `count` are updated (we denote it $\sigma_{\{\texttt{rs,count}\}}$). Hence our synthesis problem has been reduced to three independent sub-problems, $\sigma_{start} \to \sigma_{\{\texttt{rs}\}}$, $\sigma_{\{\texttt{rs}\}} \to \sigma_{\{\texttt{rs,count}\}}$, and $\sigma_{\{\texttt{rs,count}\}} \to \sigma_{end}$, each only requiring an assignment to a single variable.

Intermediate State Graph. Of course, in reality the order of assignments is not given: this is what the user needed help with in the first place! The bad news is that for a hole with n output variables there are $n!$ possible assignment orders to consider, but the good news is that the synthesizer need not enumerate them all explicitly, because different orders share intermediate states and assignment sub-sequences. For example, both the assignment order `last`, `rs`, `count` and the assignment order `last`, `count`, `rs` pass through the intermediate state $\sigma_{\{\texttt{last}\}}$, and likewise both `rs`, `last`, `count` and `last`, `rs`, `count` pass through $\sigma_{\{\texttt{rs,last}\}}$.

To take advantage of these shared states, we propose a new data structure we dub an

Intermediate State Graph (ISG). The ISG for our running example is shown in Fig. 3.4. The nodes in this graph are all the relevant states of a synthesis problem—σ_{start}, σ_{end}, and the intermediate states σ_X for all subsets $X \subset V$ of the output variables; the edges in this graph connect a state to all states with exactly one more updated variable.[6] Each ISG node except σ_{end} holds a separate bottom-up expression enumerator.

When the enumerator at the ISG node $\sigma_{\{\texttt{rs}\}}$ encounters the expression `c`, this expression gets evaluated in the state $\sigma_{\{\texttt{rs}\}}$ and tested as a possible assignment for all outgoing edges from that node. In our example, the assignment `last = c` transforms $\sigma_{\{\texttt{rs}\}}$ into $\sigma_{\{\texttt{rs},\texttt{last}\}}$, hence we label the outgoing edge `last` from $\sigma_{\{\texttt{rs}\}}$ with this program and mark the edge solved. A solution to the synthesis problem is found when there is a path from σ_{start} to σ_{end} along solved edges. For this example, we get a solution by traversing the path $\sigma_{start}, \sigma_{\{\texttt{rs}\}}, \sigma_{\{\texttt{rs},\texttt{count}\}}, \sigma_{end}$.

3.2.3 Synthesizing conditional statements

As a final challenge, assume the user now wants the synthesizer to generate the entire loop body, by inserting the same multi-variable hole immediately after the loop header and providing after-states for the first five iterations of the loop, as shown in Fig. 3.1c. Again, in less than a second, LooPy replaces the hole with the code shown on the right.

```
if c == last:
  count += 1
else:
  rs += str(count) + last
  count = 1
  last = c
```

To understand how LooPy generates this code so quickly, the final missing piece is efficient synthesis of conditionals.

Our technique for synthesizing conditionals is inspired by the divide-and-conquer approach from EUSolver [10], but is adapted to the context of ISGs. As an example, consider a simplified synthesis problem defined by the before-after pairs ε^0 and ε^1 from the first two loop

[6]For simplicity, here we ignore Python's simultaneous assignment statements, which update multiple variables at a time. In Sec. 3.4 and Sec. 3.5.3 we show how LooPy synthesizes such assignments.

iterations in Fig. 3.1c, where $\varepsilon^i = \sigma^i_{start} \to \sigma^i_{end}$ for $i = 0, 1$ and

$$\sigma^0_{start} = \{ \; \texttt{c} \mapsto \texttt{'a'}, \texttt{rs} \mapsto \texttt{''}, \; \texttt{count} \mapsto 1, \texttt{last} \mapsto \texttt{'a'} \} \qquad \sigma^0_{end} = \{ \; \texttt{c} \mapsto \texttt{'a'}, \texttt{rs} \mapsto \texttt{''}, \; \texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'} \}$$

$$\sigma^1_{start} = \{ \; \texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{''}, \; \texttt{count} \mapsto 2, \texttt{last} \mapsto \texttt{'a'} \} \qquad \sigma^1_{end} = \{ \; \texttt{c} \mapsto \texttt{'b'}, \texttt{rs} \mapsto \texttt{'2a'}, \; \texttt{count} \mapsto 1, \texttt{last} \mapsto \texttt{'b'} \}$$

The synthesizer has to decide whether to generate a single assignment sequence that satisfies *both* of these examples, or to synthesize two branches, where the first one satisfies ε^0 and the second one satisfies ε^1; more generally, with k examples, the synthesizer needs to guess the right *partitioning* of these examples into the two branches. LOOPY is able to consider all possible partitions simultaneously using the following modification to the ISG.

Consider again the ISG in Fig. 3.4, but now imagine that every node is associated with a vector of states (one for each example ε^0 and ε^1). Let us focus on the transition between σ_{start} and $\sigma_{\{\texttt{count}\}}$. Instead of a single edge between these nodes that satisfies both examples, LOOPY now constructs *two* edges, one for each partition of the example set: $\langle \{\varepsilon^0, \varepsilon^1\}, \emptyset \rangle$ and $\langle \{\varepsilon^0\}, \{\varepsilon^1\} \rangle$. The edge label now contains two separate solutions for `count`: one for the *then* branch of the conditional and one for the *else* branch. Whenever a new expression is enumerated at the node σ_{start}, it is tested as a potential solution for both of the cases on each of the two edges: for example, the expression `count + 1` is correct for ε^0 but not ε^1, so we save it on the edge $\langle \{\varepsilon^0\}, \{\varepsilon^1\} \rangle$, in the *then* case; the expression `1` is correct for ε^1 but not ε^0, so we save it on the same edge but in the *else* case.

Finally, to synthesize the guard expression for the conditional, the σ_{start} node of the ISG also accumulates boolean expressions that partition the examples into all possible partitions. To construct the synthesis result, we now require the ISG to have a path from σ_{start} to σ_{end} via edges that belong to the same partition, and a boolean expression that matches that partition.

$$
\begin{array}{lll}
\hat{p} & ::= s & \text{atomic statement} \\
 & \mid \bar{x} = \texttt{??} & \text{hole} \\
 & \mid \hat{p}\ ;\ \hat{p} & \text{sequential composition} \\
 & \mid \texttt{if}\ e :\ \hat{p}\ \texttt{else} :\ \hat{p} & \text{conditional} \\
 & \mid \texttt{for}\ x\ \texttt{in}\ e :\ \hat{p} & \text{for-loop}
\end{array}
$$

Figure 3.5. The syntax of sketches $\hat{p}$ in LooPy; programs p use the same grammar excluding holes.

$$
\textsc{Atom}\ \frac{[\![s]\!](\sigma) = \sigma'}{[s]_{\mathscr{O}}(\sigma) = \sigma \rightarrow^{s} \sigma'}
\qquad
\textsc{Hole}\ \frac{}{[h]_{\mathscr{O}}(\sigma) = \sigma \rightarrow^{h} \mathscr{O}(\sigma)}
$$

$$
\textsc{Seq}\ \frac{[\hat{p}_1]_{\mathscr{O}}(\sigma^0) = \sigma^0 \ldots \rightarrow^{\hat{s}_1} \sigma^1 \qquad [\hat{p}_2]_{\mathscr{O}}(\sigma^1) = \sigma^1 \ldots \rightarrow^{\hat{s}_2} \hat{\sigma}^2}{[\hat{p}_1\ ;\ \hat{p}_2]_{\mathscr{O}}(\sigma^0) = \sigma^0 \ldots \rightarrow^{\hat{s}_1} \sigma^1 \ldots \rightarrow^{\hat{s}_2} \hat{\sigma}^2}
$$

$$
\textsc{SeqBot}\ \frac{[\hat{p}_1]_{\mathscr{O}}(\sigma^0) = \sigma^0 \ldots \rightarrow^{h} \bot}{[\hat{p}_1\ ;\ \hat{p}_2]_{\mathscr{O}}(\sigma^0) = \sigma^0 \ldots \rightarrow^{h} \bot}
$$

Figure 3.6. Definition of a live trace $[\hat{p}]_{\mathscr{O}}(\sigma)$ of a sketch $\hat{p}$ starting from store σ using a live execution oracle $\mathscr{O}$; rules for conditionals and loops are omitted for brevity.

3.3 From Live PBE to Block-Level Synthesis

In this section, we define two forms of synthesis tasks: a *live PBE task*, specified by a LooPy user, and a *block-level synthesis task*, solved by the the LooPy synthesis engine. We also formalize live execution as the mechanism that reduces the former task to the latter.

We define our synthesis tasks over a subset of Python shown in Fig. 3.5 (for now ignore the hole statement, which can appear in sketches but not in programs). The structure of expressions e and atomic statements s is irrelevant for purposes of this section, and therefore omitted from the figure. We assume, however, that they are equipped with semantics $[\![e]\!] : \mathsf{Store} \rightarrow \mathsf{Val}$ and $[\![s]\!] : \mathsf{Store} \rightarrow \mathsf{Store}$ (where Val is the set of values and Store is the set of stores σ that map variables to values). We combine the semantics of atomic statements with the standard behavior of control structures to define a *program trace* $[p](\sigma^0) = \sigma^0 \rightarrow^{s_1} \sigma^1 \rightarrow^{s_2} \cdots \rightarrow^{s_n} \sigma^n$ as the sequences of stores σ^i and atomic statements s_i that a program execution starting at σ^0 goes through, such that $\sigma^i = [\![s_i]\!](\sigma^{i-1})$.

Sketches and live exectuion. A *sketch* $\hat{p}$ is a program with exactly one hole statement $\overline{x} = \texttt{??}$, where $\overline{x}$ denotes one or more program variables, called the *output variables* of the hole. We use the meta-variable h to range over holes and $\hat{s}$ to range over the union of atomic statements and holes.

A *live execution oracle* is a function $\mathscr{O}$ that, given a store σ, returns either a new store σ' or $\bot$. Note that the oracle need not take the hole as input since a sketch has only one hole (we do assume, however, that the oracle is specific to a sketch). We also assume that $\mathscr{O}$ only updates the output variables of the hole, *i.e.*, for a sketch with the hole $\overline{x} = \texttt{??}$ and for any store σ and program variable $y \notin \overline{x}$, $\mathscr{O}(\sigma)(y) = \sigma(y)$.

Using the oracle, we define a *live trace* $[\hat{p}]_{\mathscr{O}}(\sigma^0)$ of a sketch $\hat{p}$ starting in the store σ^0. A live trace is a sequence $\sigma^0 \rightarrow^{\hat{s}_1} \sigma^1 \rightarrow^{\hat{s}_2} \cdots \rightarrow^{\hat{s}_n} \hat{\sigma}^n$, where the last state $\hat{\sigma}^n$ can be either a store σ^n or $\bot$. Live traces are defined using rules in Fig. 3.6. The differences from regular program traces are captured in the rules HOLE, which uses the oracle to execute a hole statement, and SEQBOT, which *suspends* live execution once the oracle returns $\bot$ (a similar suspension happens in the rule for loops, which is omitted for brevity). We say that a program trace t *refines* a live trace $\hat{t}$, written $t \prec \hat{t}$, if t can be obtained from $\hat{t}$ by replacing every step of the form $\sigma \rightarrow^{h} \hat{\sigma}'$ with a trace $\sigma \rightarrow^{s_1} \ldots \rightarrow^{s_n} \sigma''$, where either $\hat{\sigma}' = \sigma''$ or $\hat{\sigma}' = \bot$. In other words, t passes through the same stores as $\hat{t}$, except that oracle steps can be replaced with multiple atomic steps, and if $\hat{t}$ was suspended, t instead continues execution.

With these preliminaries, we can define the live PBE task:

Definition 1 (Live PBE task)**.** A *live PBE task* is a triple $\langle \hat{p}, \mathscr{O}, \sigma^0 \rangle$ of a sketch $\hat{p}$, a live execution oracle $\mathscr{O}$, and an initial store σ^0. A *solution* to a synthesis task is a program p such that

1) $\exists p^*. p = \hat{p}[p^*/h]$, *i.e.*, p is the sketch with its hole replaced by some program p^*; and

2) $[p](\sigma^0) \prec [\hat{p}]_{\mathscr{O}}(\sigma^0)$, *i.e.*, the execution trace of p refines the live trace of the sketch.

The two conditions above capture the *syntactic* and *semantic* expectations of a live PBE user, respectively. The first condition prevents the synthesizer from replacing any part of the sketch other than the hole. The second condition requires the synthesized program to behave like the live execution for as long as possible (until the point where the latter was suspended). For simplicity, we define a live PBE task using a single initial store σ^0; the definition can be easily generalized to multiple initial stores; note, however, that if the hole is inside a loop, even a single initial store can lead to multiple occurrences of the hole in the live trace, and hence multiple invocations of the oracle.

The* LOOPY *UI. In LOOPY, the user provides the sketch $\hat{p}$ and the initial store σ^0 (see *e.g.*, the call `compress('aabccca')` in line 12 in Fig. 3.3), and also serves as the live execution oracle $\mathscr{O}$. The LOOPY UI incrementally builds the live trace $[\hat{p}]_{\mathscr{O}}(\sigma^0)$ by querying the oracle via projection boxes, as shown in Fig. 3.2. More precisely, LOOPY first builds the prefix $\sigma^0 \cdots \rightarrow^{s_1} \sigma^1 \rightarrow^{h}$ of the live trace until it first encounters the hole; it then displays the store σ^1 in the left-hand side of the projection box, and prompts the user to enter new values for the output variables of h in the right-hand side. If the user presses "enter", $\mathscr{O}(\sigma^1)$ is taken to be $\perp$, and the live trace is complete; otherwise execution continues with the updated store until it either terminates or encounters the hole again (in a later loop iteration), which triggers another query to the oracle (in the next line of the projection box).

Block-level synthesis. Given a live PBE task $\langle \hat{p}, \mathscr{O}, \sigma^0 \rangle$, we can now use its live trace, generated by the LOOPY UI, to derive a simpler, *local* specification for $\hat{p}$'s hole h, which we refer to as the *block-level synthesis task*. The LOOPY synthesizer can then simply solve this local task, ignoring the fact that the original sketch might have contained a loop.

Definition 2 (Block-level synthesis task)**.** A block-level synthesis task is defined by the set of examples $\mathscr{E}$, where each example ε is a pair of stores $\sigma_{start} \rightarrow \sigma_{end}$. A *solution* to a block-level task is a program p^* with no holes or loops, such that for every $\sigma_{start} \rightarrow \sigma_{end} \in \mathscr{E}$, $[p^*](\sigma_{start}) = \sigma_{start} \ldots \rightarrow^{s} \sigma_{end}$.

To reduce the live PBE task $\langle \hat{p}, \mathscr{O}, \sigma^0 \rangle$ to a block-level task, we define $\mathscr{E} = \{\sigma \to \sigma' \mid \sigma \to^h \sigma' \in [\hat{p}]_{\mathscr{O}}(\sigma^0)\}$. It is easy to show by comparing the definitions of the two synthesis tasks, that if p^* is a solution to the block-level task $\mathscr{E}$, then $\hat{p}[p^*/h]$ is a solution to original live PBE task $\langle \hat{p}, \mathscr{O}, \sigma^0 \rangle$.

The following two sections will address the synthesis of loop-free blocks of code as a solution to the block-level synthesis task. For the sake of presentation, Sec. 3.4 focuses on straight-line code blocks (sequences of assignments); then Sec. 3.5 extends the synthesis algorithm to conditionals.

3.4 Synthesizing Sequences of Assignments

In this section, we describe LOOPY's block-level synthesis algorithm for straight-line programs, *i.e.*, when the solution p^* is drawn from the following grammar:

$$p \ ::= \ \bar{x} = \bar{e} \ \mid \ p \, ; \, p$$

Here $\bar{x} = \bar{e}$ is Python's *simultaneous assignment* statement, which has one or more variables on the left and the same number of expressions on the right. The semantics of a simultaneous assignment is to first evaluate each e_i in the current store and then to assign its value to the corresponding x_i. For example, executing `x, y = x + x, x + x + x` in the store $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}$ yields the store $\{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 3\}$.

We begin our exposition in Sec. 3.4.1 with the simplest version of the algorithm, where the synthesis task is restricted to a single example ($|\mathscr{E}| = 1$), and the solution is restricted to a single (simultaneous) assignment. Sec. 3.4.2 then extends the algorithm to generate a sequence of assignments, and Sec. 3.4.3 extends it to handle multiple examples.

Expression enumerators. All variants of LOOPY's synthesis algorithm require enumerating expressions e, used on the right-hand side of assignments (and in the guards of conditionals later on). To this end, LOOPY relies on an existing algorithm called *bottom-up enumeration* *with Observational Equivalence reduction* [177, 6]. Bottom-up enumeration gradually builds up a bank of larger and larger expressions, by combining sub-expressions that are already in the bank. Observational Equivalence (OE) speeds up this process by evaluating every expression on a given set of inputs and only retaining one expression per result in the bank.

For the purposes of Sec. 3.4.1–3.4.2, we can think of an expression enumerator as a black-box function $\mathsf{Enum}(\sigma)$, which is parameterized by a store,[7] and produces a (possibly infinite) stream of expressions $e_0, e_1, \ldots$. The reason an enumerator takes σ as a parameter is twofold: first, it uses the store's domain, $\mathsf{Vars}(\sigma)$, as the set of free variables in the expressions it builds; second, it uses σ to evaluate the expressions for the purposes of OE reduction. OE reduction guarantees that the enumerator $\mathsf{Enum}(\sigma)$ is *complete on* σ, that is, for any value v, if there exists an expression e such that $[\![e]\!](\sigma) = v$, then a (possibly different) expression e_i such that $[\![e_i]\!](\sigma) = v$ will eventually be enumerated; in other words, OE discards programs but never discards distinct evaluation results on σ.

When LOOPY solves a synthesis task $\sigma_{start} \to \sigma_{end}$ and uses an enumerator to synthesize a sub-expression e of a program p, it is crucial that the enumerator be initialized with the store σ *in which e will be evaluated* during the execution of p on σ_{start}. Depending on where e is located inside p, σ might or might not be equal to σ_{start}. Passing a wrong store to the enumerator leads to incompleteness: we can no longer assume that if an expression with the required value exists, it will be enumerated. For this reason, LOOPY often needs to create multiple enumerators with different stores.

3.4.1 Single Example, Single Assignment

We begin by examining the case where $\mathscr{E} = \{\varepsilon\}$ and the target program contains a single assignment. We will denote Mod_ε the set of variables *modified* by an example $\varepsilon = \sigma_{start} \to \sigma_{end}$, *i.e.*, those variables x where $\sigma_{end}(x) \neq \sigma_{start}(x)$. The solution to our block-level synthesis task is hence of the form $\overline{x} = \overline{e}$, where the left-hand side contains exactly the variables in Mod_ε.

[7]In Sec. 3.4.3 we will generalize the notion of expression enumerators from a single store to a vector of stores.

Synthesis algorithm. The synthesis algorithm maintains a *variable assignment* $\mathscr{A}$, which maps each variable in Mod_ε to an expression or $\bot$; the assignment is initialized by setting $\mathscr{A}(x) = \bot$ for every $x \in \mathsf{Mod}_\varepsilon$. The algorithm then creates a single expression enumerator $\mathsf{Enum}(\sigma_{start})$. In each iteration, it draws another expression e_i from the enumerator stream, and for every unassigned variable x (such that $\mathscr{A}(x) = \bot$), it tests whether e_i is *valid* for x, *i.e.*, whether $[\![e_i]\!](\sigma_{start}) = \sigma_{end}(x)$; if so, the algorithm updates $\mathscr{A}(x)$ to e_i. When $\mathscr{A}$ is *complete* (*i.e.*, $\mathscr{A}(x) \neq \bot$ for every x), the algorithm terminates and returns $\bar{x} = \overline{\mathscr{A}(x)}$ as the solution to the block-level synthesis task. Note that a single expression enumerator is sufficient in this case because all right-hand sides of the simultaneous assignment $\bar{x} = \bar{e}$ are evaluated in the same store σ_{start}.

Example 1. Consider the specification $\sigma_{start} = \{\mathtt{x} \mapsto \mathtt{1}, \mathtt{y} \mapsto \mathtt{1}\}$ and $\sigma_{end} = \{\mathtt{x} \mapsto \mathtt{2}, \mathtt{y} \mapsto \mathtt{3}\}$, making $\mathsf{Mod}_\varepsilon = \{\mathtt{x}, \mathtt{y}\}$. We first initialize $\mathscr{A} = \{\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot\}$ and create an enumerator $\mathsf{Enum}(\{\mathtt{x} \mapsto \mathtt{1}, \mathtt{y} \mapsto \mathtt{1}\})$. After several iterations, the enumerator yields the expression `x + x`, which evaluates to `2`. This matches $\sigma_{end}(\mathtt{x})$, so we set $\mathscr{A}(\mathtt{x})$ to `x + x`. Next, the enumerator yields `x + x + x`, which evaluates to `3`, matching $\sigma_{end}(\mathtt{y})$. At this point, $\mathscr{A} = \{\mathtt{x} \mapsto \mathtt{x\ +\ x}, \mathtt{y} \mapsto \mathtt{x\ +\ x\ +\ x}\}$ is complete, so the algorithm terminates and returns the simultaneous assignment statement:

```
  x, y = x + x, x + x + x
```

3.4.2 Single Example, Sequence of Assignments

The synthesis task described above has an even simpler solution if instead of a single simultaneous assignment we perform two assignments *in sequence*: `x = x + x; y = x + y`. We can synthesize this solution by *decomposing* the overall synthesis task into two single-assignment sub-tasks: $\sigma_{start} \to \sigma_{\{\mathtt{x}\}}$, which transforms the start state into an *intermediate state* where only `x` has been updated, and $\sigma_{\{\mathtt{x}\}} \to \sigma_{end}$, which transforms the intermediate state into the end state. Each sub-task can then be solved independently using the algorithm from

Sec. 3.4.1. Since the order and grouping of assignments in the solution p^* is not known a-priori, the algorithm has to consider decomposing the problem using *all* intermediate states that p^* could possibly pass through. If we assume that p^* assigns each variable only once, there is exactly one intermediate state for each non-empty, strict subset of Mod_ε. Formally, for each $X \subset \mathsf{Mod}_\varepsilon$, let a *partially-updated state* σ_X be the state where only those variables in X have been updated:

$$\sigma_X \ = \ \{x \mapsto \sigma_{end}(x) \mid x \in X\} \cup \{x \mapsto \sigma_{start}(x) \mid x \in \mathsf{Mod}_\varepsilon \setminus X\}$$

Note that $\sigma_\emptyset = \sigma_{start}$ and $\sigma_{\mathsf{Mod}_\varepsilon} = \sigma_{end}$, and all other partially-updated states are intermediate states.

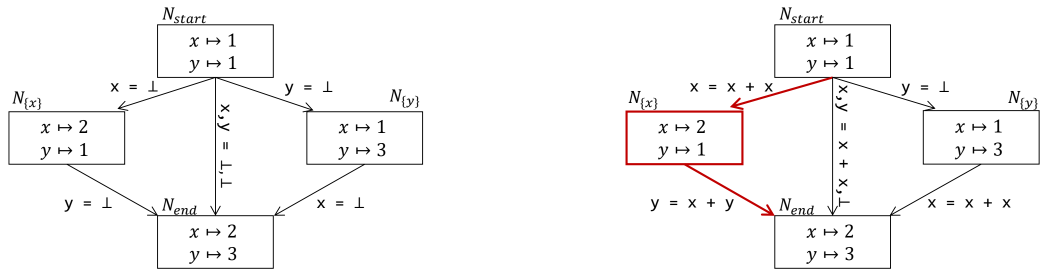
Example 2. In Ex. 1, there are two possible intermediate states to consider: $\sigma_{\{\mathtt{x}\}} = \{x \mapsto 2, y \mapsto 1\}$ and $\sigma_{\{\mathtt{y}\}} = \{x \mapsto 1, y \mapsto 3\}$. A solution p^* can transition from σ_{start} to σ_{end} through $\sigma_{\{\mathtt{x}\}}$, where `x` is modified first, through $\sigma_{\{\mathtt{y}\}}$, where `y` is modified first, or directly, in a simultaneous assignment.

Intermediate State Graph. We can compactly represent the space of all possible solutions to a synthesis task using a DAG whose nodes are partially-updated states and whose edges are single-assignment synthesis sub-tasks. We dub this data structure an *Intermediate State Graph.*

Definition 3 (Intermediate State Graph (ISG))**.** Given a synthesis task $\{\varepsilon\} = \{\sigma_{start} \to \sigma_{end}\}$, its ISG is a directed acyclic graph where:

1. there is a node N_X for each $X \subset \mathsf{Mod}\varepsilon$, which represents the partially-updated state σ_X;
2. there is an edge $(N_X, N_{X'})$ iff $X \subsetneq X'$;
3. each edge $(N_X, N_{X'})$ is labeled with a variable assignment $\mathscr{A}_{(X,X')}$, whose domain is $X' \setminus X$.

Example 3. Fig. 3.7 depicts the ISG for Ex. 1 with different variable assignments $\mathscr{A}_E$ on the edges: on the left all the assignments are empty; on the right, some (but not all) the assignments are complete.



(a) The initial ISG **(b)** The ISG with a solution

Figure 3.7. The Intermediate State Graph for Ex. 1. The path of complete edges is marked in red.

Synthesis algroithm. The synthesis algorithm maintains an ISG, where the assignment for each edge E is initialized with $\mathscr{A}_E(x) = \bot$ for every x in its domain. The algorithm then creates an expression enumerator $\mathsf{Enum}(\sigma_X)$ for each ISG node N_X except the final node N_{end}. In each iteration, the algorithm draws an expression e_i from an enumerator at some node N_X. For every outgoing edge $(N_X, N_{X'})$ and for every unassigned variable x on that edge (such that $\mathscr{A}_{(X,X')}(x) = \bot$), it tests whether e_i is *valid* for x on that edge, *i.e.*, whether $[\![e_i]\!](\sigma_X) = \sigma_{X'}(x)$; if so, the algorithm updates $\mathscr{A}_{(X,X')}(x)$ to e_i. When all variables on an edge are assigned (*i.e.*, $\mathscr{A}_{(X,X')}(x) \neq \bot$ for every x), the edge $(N_X, N_{X'})$ is marked *complete*. The algorithm terminates when there is a path from N_{start} to N_{end} via complete edges. The sequence of assignments along the path is the solution to the synthesis problem.

Example 4. To solve the synthesis problem defined in Ex. 1, we first initialize the ISG as shown in Fig. 3.7a. We then create three expression enumerators, one each in N_{start}, $N_{\{x\}}$, and $N_{\{y\}}$.

Consider the iteration of the algorithm where we query the enumerator N_{start}, and it yields the expression `x + x`. This expression, which evaluates to $[\![\texttt{x + x}]\!](\sigma_{start}) = 2$, is then tested for validity for every variable on each of the three outgoing edges from N_{start}:

- $(N_{start}, N_{\{x\}})$, variable x: $\sigma_{\{\mathrm{x}\}}(x) = 2$, so $\mathscr{A}_{(start,\{x\})}(x)$ is set to `x + x`.
- $(N_{start}, N_{\{y\}})$, variable y, $\sigma_{\{\mathrm{y}\}}(y) \neq 2$, so $\mathscr{A}_{(start,\{y\})}(y)$ remains $\bot$.
- (N_{start}, N_{end}), variables x and y: $\sigma_{end}(x) = 2$ but $\sigma_{end}(y) \neq 2$, so $\mathscr{A}_{(start,end)}(x)$ is set to `x + x`, and $\mathscr{A}_{(start,end)}(y)$ remains $\bot$.

After this iteration, the edge $(N_{start}, N_{\{x\}})$ is complete, but the two other outgoing edges are not.

At a later iteration, we encounter the expression `x + y` at a different node, $N_{\{x\}}$. This expression, which evaluates to $[\![\texttt{x + y}]\!](\sigma_{\{\texttt{x}\}}) = 3$ is tested for validity for the variable y of the sole outgoing edge $(N_{\{x\}}, N_{end})$ from $N_{\{x\}}$. Since $\sigma_{end}(y) = 3$, $\mathscr{A}_{(\{x\},end)}(y)$ is set to `x + y`, and this edge is now complete. Moreover, as shown in Fig. 3.7b, there is now a path of complete edges from N_{start} to N_{end}, which is translated into the solution `x = x + x; y = x + y`.

Multiple enumerators. As we alluded to at the beginning of this section, our synthesis algorithm for a sequence of assignments needs to use multiple enumerators, initialized with different stores σ_X, because the synthesized expressions are meant to be evaluated in different stores during program execution. To illustrate potential completeness issues when using a wrong store, assume that the ISG in Fig. 3.7 had a single shared enumerator $\mathsf{Enum}(\sigma_{start})$. Note that $[\![x]\!](\sigma_{start}) = [\![y]\!](\sigma_{start}) = 1$, so from the standpoint of $\mathsf{Enum}(\sigma_{start})$ these two expressions are equivalent, and one of them (say y) is discarded by OE. As a result $\mathsf{Enum}(\sigma_{start})$ will never enumerate any expressions with variable y; in particular, using just this enumerator, we would not be able to generate the desired solution `x = x + x; y = x + y`. Instead, the enumerator $\mathsf{Enum}(\sigma_{\{\texttt{x}\}})$ yields both x and y (and larger expressions build from both of these variables), since the two variables are not equivalent in $\sigma_{\{\texttt{x}\}}$.

Design considerations. A shrewd reader might be concerned that the enumerators at different nodes duplicate each other's work, since they do enumerate some of the same expressions. An alternative design that eliminates this work duplication is to use a single *shared enumerator* initialized with a vector of all partially updated states: $\mathsf{Enum}_{\overline{\sigma_X}}$. This enumerator treats each σ_X as a separate example it must consider, and prunes expressions by evaluating them point-wise on $\overline{\sigma_X}$ and comparing their output vectors. Indeed, we can safely share this single enumerator between all nodes in the ISG, without the danger of any relevant expressions being lost, as all states on which the expressions might possibly be evaluated are taken into account. Perhaps surprisingly, this design turned out to be so inefficient, that it did not even warrant a quantitative evaluation. The reason is that the shared enumerator has a much more fine-grained equivalence relation between expressions, which prevents OE reduction from pruning expressions efficiently; as a result, the shared enumerator produces many more expressions than all per-node enumerators combined.

The algorithm above does not specify the order in which different enumerators are queried. Our current implementation interleaves them in a round-robin fashion. In principle they could easily run in parallel, as the only operation that requires coordination between multiple ISG nodes is checking for a complete path. As long as the scheduling of enumerators is starvation-free, it does not affect the soundness or completeness of the algorithm, although it can affect the size of the generated solution.

Picking the smallest program. Because a single edge can be shared by multiple paths, the synthesis algorithm might discover multiple complete paths from N_{start} to N_{end} in a single iteration. In this case LOOPY selects the program with the smallest total size. To that end, each edge $(N_X, N_{X'})$ is assigned a weight equal to $\sum_{x \in X' \setminus X} \mathsf{size}(\mathscr{A}_{(X,X')}(x))$, where size is the size of an expression in AST nodes (with $\mathsf{size}(\bot) = \infty$). LOOPY then uses Dijkstra's algorithm to find the shortest path from N_{start} to N_{end}.

In general, the synthesis algorithm is not guaranteed to find the smallest solution overall, because the size of expressions enumerated at different nodes increases at a different rate (*e.g.*, in Fig. 3.7 N_{start} starts producing larger expressions sooner than the other nodes, since it only has to consider expressions with a single free variable)[8]; our experiments show, however, that the round-robin interleaving produces small programs in practice.

3.4.3 Multiple Examples, Sequence of Assignments

Recall that in Live PBE, a hole inside a loop typically generates a block-level synthesis task with multiple examples (one for each loop iteration entered by the user). We now extend

[8]It is theoretically possible to pause each enumerator that finishes program size k until all enumerators finish size k, and start size $k+1$ together. This is impractical, however, since the set of programs of size k can be very large even for a moderate k.

our synthesis algorithm to this setting. More specifically, let $\mathscr{E} = \langle \varepsilon^1, \ldots, \varepsilon^n \rangle$ be a *vector* of examples (the ordering of examples is irrelevant, but fixed once and for all before synthesis begins). We define the set of modified variables $\mathsf{Mod}_{\mathscr{E}}$ to include variables modified in any of the examples: $\mathsf{Mod}_{\mathscr{E}} = \bigcup_{\varepsilon \in \mathscr{E}} \mathsf{Mod}_{\varepsilon}$.

Expression enumerators. An expression enumerator $\mathsf{Enum}(\langle \sigma^1, \ldots, \sigma^n \rangle)$ must now be parametrized by a vector of stores. Each expression e_i it constructs is evaluated in all stores to produce an *output vector* $\langle v^1, \ldots, v^n \rangle$; output vectors are compared point-wise, and only one expression per vector is retained in the bank. As usual with OE reduction, the more examples we have, the more expressions are retained in the bank, and the slower the enumeration.

Intermediate State Graph. The only change in the ISG is that a node N_X now corresponds to a *vector* of partially-updated stores, rather than a single store:

$$N_X = \langle \sigma_X^1, \ldots \sigma_X^n \rangle \quad \text{where } \sigma_X^k = \{x \mapsto \sigma_{end}^k(x) \mid x \in X\} \cup \{x \mapsto \sigma_{start}^k(x) \mid x \in \mathsf{Mod}_{\mathscr{E}} \setminus X\}$$

Importantly, the topology of the graph is unchanged, in the sense that the set of nodes and edges is determined only by $\mathsf{Mod}_{\mathscr{E}}$ and does not directly depend on n.

Synthesis algorithm. The synthesis algorithm remains largely the same. The expression enumerator is each node N_X is now $\mathsf{Enum}(\langle \sigma_X^1, \ldots \sigma_X^n \rangle)$. An expression e is valid for a variable x on an edge $(N_X, N_{X'})$ if it is valid *for every example*: $\forall 1 \leq k \leq n. [\![e]\!](\sigma_X^k) = \sigma_{X'}^k(x)$.

In the next section, we tackle synthesis of conditionals, which requires multiple examples; hence from now on we use these generalized versions of the ISG and the synthesis algorithm.

3.5 Synthesizing Conditionals

In this section, we extend our block-level synthesis algorithm to support conditional statements. We begin in Sec. 3.5.1 with a restricted setting where the solution has a single assignment in each branch of the conditional; Sec. 3.5.2 extends the algorithm to combine conditionals with assignment sequences.

3.5.1 Single Conditional Assignment

Consider a block-level synthesis task $\mathscr{E}$ with multiple examples ($|\mathscr{E}| > 1$) and a single modified variable x ($\mathsf{Mod}_{\mathscr{E}} = \{x\}$), and assume that we are looking for a solution p^* of the form:

$$\texttt{if } e_{cond} : \; x = e_{then} \texttt{ else} : \; x = e_{else}$$

Unlike an unconditional assignment $x = e$, where we had to find a single expression e that is valid *for all* examples $\mathscr{E}$, in this case we have to find three expressions, e_{cond}, e_{then}, and e_{else} such that:

1. e_{then} is valid for some subset of examples $\mathscr{E}_{then} \subset \mathscr{E}$;
2. e_{else} is valid for the rest of the example $\mathscr{E}_{else} = \mathscr{E} \setminus \mathscr{E}_{then}$; and
3. e_{cond} is a boolean expression that separates these two subsets, *i.e.*, evaluates to `True` on all $\mathscr{E}_{then}$ and to `False` on all $\mathscr{E}_{else}$.

The general idea behind the synthesis algorithm is to use a single enumerator $\mathsf{Enum}(\langle\sigma^1_{start}, \ldots, \sigma^n_{start}\rangle)$ to generate a stream of expressions, and keep track of promising candidates for e_{then}, e_{else}, and e_{cond}, until we have encountered three expressions that together satisfy the above requirements.

Partitions. Since we do not know in advance how to partition $\mathscr{E}$ into $\mathscr{E}_{then}$ and $\mathscr{E}_{else}$, the algorithm must consider all possible partitions $\pi = \langle\mathscr{E}_1, \mathscr{E}_2\rangle$. Note, however, that a solution for $\langle\mathscr{E}_1, \mathscr{E}_2\rangle$, can be transformed into a solution for the symmetric partition $\langle\mathscr{E}_2, \mathscr{E}_1\rangle$, by swapping e_{then} and e_{else} and negating e_{cond}. Hence the algorithm only needs to explicitly track half of all partitions (modulo symmetry); we denote this set of partitions of interest $\Pi_{\mathscr{E}}$, with $|\Pi_{\mathscr{E}}| = 2^{|\mathscr{E}|-1}$.

Example 5. Consider a synthesis task $\mathscr{E} = \{\varepsilon^1, \varepsilon^2\}$ where $\varepsilon^1 = \{x \mapsto -1\} \to \{x \mapsto 1\}$ and

$\varepsilon^2 = \{x \mapsto 2\} \to \{x \mapsto 2\}$. There are four partitions of $\mathscr{E}$:

$$\pi_1 = \langle \mathscr{E}, \emptyset \rangle \qquad \pi_3 = \langle \{\varepsilon^2\}, \{\varepsilon^1\} \rangle$$
$$\pi_2 = \langle \{\varepsilon^1\}, \{\varepsilon^2\} \rangle \qquad \pi_4 = \langle \emptyset, \mathscr{E} \rangle$$

For the purposes of synthesis, π_2 and π_3 are symmetric, and so are π_1 and π_4 (the latter pair corresponds to an unconditional solution). Hence we select $\Pi_{\mathscr{E}} = \{\pi_1, \pi_2\}$.

Storing candidate expressions. For each partition $\langle \mathscr{E}_1, \mathscr{E}_2 \rangle \in \Pi_{\mathscr{E}}$, the algorithm maintains a pair of variable assignments, $\langle \mathscr{A}^{\mathscr{E}_1}, \mathscr{A}^{\mathscr{E}_2} \rangle$. An assignment $\mathscr{A}^{\mathscr{E}_j}$ maps each variable in $\mathsf{Mod}_{\mathscr{E}}$ (in this subsection, just x) to an expression e that is *valid* over $\mathscr{E}_j$: that is, $\forall \sigma_{start} \to \sigma_{end} \in \mathscr{E}_j . [\![e]\!](\sigma_{start}) = \sigma_{end}(x)$. The expressions stored in $\mathscr{A}^{\mathscr{E}_1}$ and $\mathscr{A}^{\mathscr{E}_2}$ are used as candidates for e_{then} and e_{else}, respectively. Note that in total there is one variable assignment for each subset of $\mathscr{E}$.

In addition to the $2^{|\mathscr{E}|}$ variables assignments, the algorithm also maintains a single *condition store* $\mathscr{C}$, which maps every partition $\langle \mathscr{E}_1, \mathscr{E}_2 \rangle \in \Pi_{\mathscr{E}}$ to a boolean expression b that *matches* this partition: that is, evaluates to `True` on $\mathscr{E}_1$ ($\forall \sigma_{start} \to \sigma_{end} \in \mathscr{E}_1 . [\![b]\!](\sigma_{start}) = \texttt{True}$) and to `False` on $\mathscr{E}_2$ ($\forall \sigma_{start} \to \sigma_{end} \in \mathscr{E}_2 . [\![b]\!](\sigma_{start}) = \texttt{False}$). The expressions stored in $\mathscr{C}$ are used as candidates for e_{cond}. Both $\mathscr{A}^{\mathscr{E}_i}$ and $\mathscr{C}$ are partial maps, *i.e.*, some of their keys may be mapped to $\bot$.

Example 6. Fig. 3.8 shows two different states of the synthesis algorithm for the task from Ex. 5. Each state is depicted as a degenerate ISG with only two nodes—N_{start} and N_{end}—since in this subsection we are not dealing with sequences of assignments. Instead of a single edge connecting the two nodes, there are now two: one edge per partition $\pi \in \Pi_{\mathscr{E}}$. Each edge is labeled with the pair of assignments associated with its partition. The node N_{start} is also labeled with the condition store, which stores a boolean expression for each of the two partitions.

Synthesis algorithm. The algorithm begins by initializing all $\mathscr{A}^{\mathscr{E}_j}(x)$ and $\mathscr{C}(\pi)$ to $\bot$, except

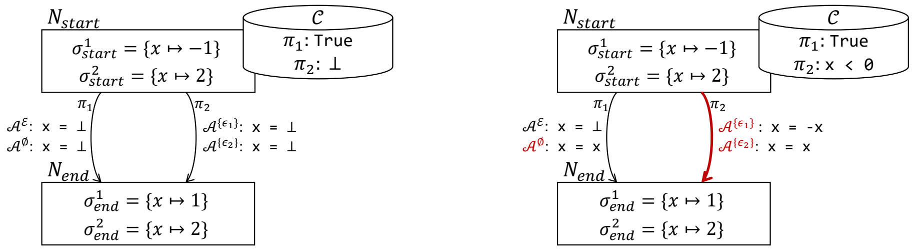

N_{start}
$\sigma^1_{start} = \{x \mapsto -1\}$
$\sigma^2_{start} = \{x \mapsto 2\}$
$\mathcal{C}$
π_1: True
π_2: ⊥
$\mathcal{A}^{\mathcal{E}}$: x = ⊥
$\mathcal{A}^{\emptyset}$: x = ⊥
$\mathcal{A}^{\{\epsilon_1\}}$: x = ⊥
$\mathcal{A}^{\{\epsilon_2\}}$: x = ⊥
N_{end}
$\sigma^1_{end} = \{x \mapsto 1\}$
$\sigma^2_{end} = \{x \mapsto 2\}$
π_2: x < 0
$\mathcal{A}^{\mathcal{E}}$: x = ⊥
$\mathcal{A}^{\emptyset}$: x = x
$\mathcal{A}^{\{\epsilon_1\}}$: x = -x
$\mathcal{A}^{\{\epsilon_2\}}$: x = x

Figure 3.8. Initial (left) and final (right) state of the synthesis algorithm for the task in Ex. 5. Complete variable assignments and complete edges (partitions) are highlighted in red. On the right, partition π_2 is complete, because both of its variable assignments are complete, and it has a condition in $\mathscr{C}$.

$\mathscr{C}(\langle \mathscr{E}, \emptyset \rangle) \mapsto$ `True`. In each iteration, the algorithm draws one expression e_i from the enumerator and updates the state as follows:

1. For each partition $\langle \mathscr{E}_1, \mathscr{E}_2 \rangle \in \Pi_{\mathscr{E}}$, the algorithm tests whether e_i is valid over either of the $\mathscr{E}_j$; in that case, $\mathscr{A}^{\mathscr{E}_j}(x)$ is updated to e^i unless already set to something other than $\perp$.
2. If e_i is a boolean expression that evaluates without errors on all examples, the algorithm searches for a partition $\pi \in \Pi_{\mathscr{E}}$ such that e_i matches π; if found, $\mathscr{C}(\pi)$ is updated to e_i, unless already set. Note that a matching partition π might not exist in $\Pi_{\mathscr{E}}$, since $\Pi_{\mathscr{E}}$ only stores half of the partitions; in this case, there must be a $\pi' \in \Pi_{\mathscr{E}}$ that is symmetric with π, and moreover the expression `not` e_i matches π'. Hence, if a matching partition for e_i is not found, then the algorithm searches for one for `not` e_i, and updates $\mathscr{C}$ accordingly.

The algorithm terminates as soon as some partition $\pi^* = \langle \mathscr{E}_1^*, \mathscr{E}_2^* \rangle$ is *complete*, that is, $\mathscr{C}(\pi^*) \neq \perp$ and both $\mathscr{A}^{\mathscr{E}_1^*}$ and $\mathscr{A}^{\mathscr{E}_2^*}$ are complete (do not contain $\perp$). The algorithm returns a solution where:

$$e_{cond} = \mathscr{C}(\pi^*) \quad e_{then} = \mathscr{A}^{\mathscr{E}_1^*}(x) \quad e_{else} = \mathscr{A}^{\mathscr{E}_2^*}(x)$$

Example 7. For the synthesis task in Ex. 5, the state is initialized as shown in Fig. 3.8 (left). The first iteration enumerates expression `x`, which is valid for the example ε^2, and hence we update $\mathscr{A}^{\{\varepsilon^2\}}(\mathtt{x}) = \mathtt{x}$ (and also, trivially, $\mathscr{A}^{\emptyset}(\mathtt{x}) = \mathtt{x}$). A later iteration enumerates `-x`, which is valid for ε^1, updating $\mathscr{A}^{\{\varepsilon^1\}}(\mathtt{x}) = \mathtt{-x}$. At this point, the partition $\pi_2 = \langle \{\varepsilon^1\}, \{\varepsilon^2\} \rangle$ has both of its variable assignments complete, but the partition itself is not yet complete, since it does not have a condition in $\mathscr{C}$. Once the enumerator yields the expression `x < 0`, we notice that it evaluates to `True` on ε^1 and to `False` on ε^2, hence we update $\mathscr{C}(\pi_2) =$ `x < 0`. At this point, π_2 is complete, and the algorithm terminates. This final state is depicted in Fig. 3.8 (right); the highlighted partition creates the final solution:

```
if x < 0:  x = -x else:  x = x
```

Multiple solutions. Because expressions e_{then} and e_{else} might be valid on overlapping subsets of examples, our algorithm may complete multiple partitions in the same iteration. Just like in Sec. 3.4.2, we pick the solution with the smallest overall size in AST nodes, *i.e.*, minimizing $\mathsf{size}(e_{cond}) + \mathsf{size}(e_{then}) + \mathsf{size}(e_{else})$. For the partition $\langle \mathscr{E}, \emptyset \rangle$, the size is computed simply as $\mathsf{size}(e_{then})$, since this solution can be simplified into an unconditional assignment.

3.5.2 Conditionals and Assignment Sequences

Finally, we present our block-level synthesis algorithm in all generality, combining the notion of an ISG from Sec. 3.4.2 to handle sequential composition with partitions from Sec. 3.5.1 to handle conditionals. In theory this approach can support programs with arbitrary combinations of assignments, conditionals, and sequential composition, drawn from the grammar:

$$p \ ::= \ \overline{x} = \overline{e} \ \mid \ p\,;\,p \ \mid \ \texttt{if}\ e:\ p\ \texttt{else:}\ p$$

In practice, however, the full search space turns out to be too large, slowing down the search and leaving the user to weed through many irrelevant solutions. Hence, the version of the algorithm implemented in LOOPY and described in this section restricts the search space to programs with a single top-level conditional, with a sequence of assignments in each branch:

$$
\begin{aligned}
p &::= q \mid \texttt{if}\, e:\ q\, \texttt{else:}\ q \\
q &::= \bar{x} = \bar{e} \mid q\,;\, q
\end{aligned}
$$

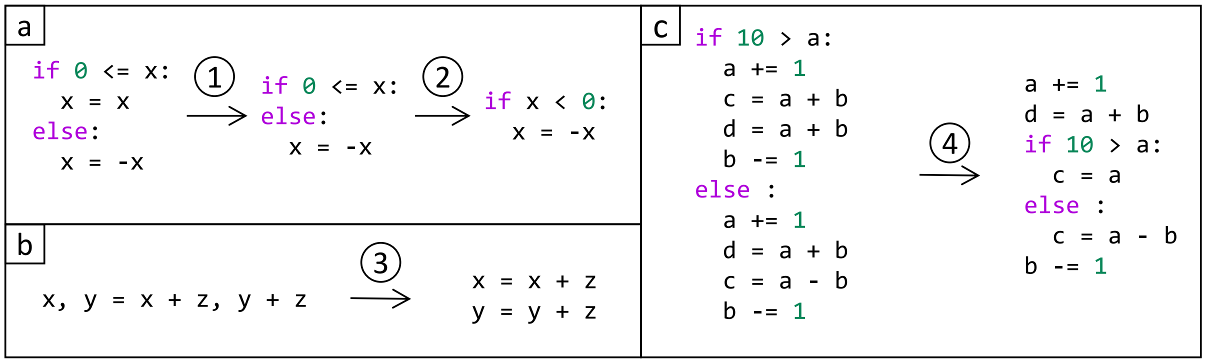
Conditional ISG. To represent programs from this space, we generalize the notion of an ISG from Def. 3 into a *conditional ISG*. We have already seen a simple conditional ISG with just two nodes in Fig. 3.8. In general, to turn an ISG into a conditional ISG we simply clone every edge $2^{|\mathscr{E}|-1}$ times (one for each partition), and associate two variable assignments (instead of one) with each edge. We also add a single condition store $\mathscr{C}$, associated with the node N_{start}.

Definition 4 (Conditional ISG)**.** Given a synthesis task $\mathscr{E} = \{\varepsilon^1, \dots, \varepsilon^n\}$, its conditional ISG is a directed acyclic *multi-graph* where:

1. there is a node N_X for each $X \subset \mathsf{Mod}_{\mathscr{E}}$, which represents the vector of partially-updated stores $\langle \sigma_X^1, \dots, \sigma_X^n \rangle$;
2. for each pair of states $N_X, N_{X'}$ such that $X \subsetneq X'$, and for each partition $\pi \in \Pi_{\mathscr{E}}$, there is an edge $(N_X, N_{X'})_\pi$;
3. each edge $(N_X, N_{X'})_\pi$ where $\pi = \langle \mathscr{E}_1, \mathscr{E}_2 \rangle$ is labeled with a pair of variable assignments $\mathscr{A}^{\mathscr{E}_1}_{(X,X')}$ and $\mathscr{A}^{\mathscr{E}_2}_{(X,X')}$, whose domain is $X' \setminus X$.
4. the node N_{start} is labeled with a condition store $\mathscr{C}$.

Given a conditional ISG $\mathscr{G}$ and a partition π, we denote $\mathscr{G}/\pi$ the sub-graph of $\mathscr{G}$ spanned by all the edges of the form $(N, N')_\pi$. Each $\mathscr{G}/\pi$ is a regular DAG (not a multi-graph), which represents the current candidate solution for partition π.

Synthesis algorithm. The high-level structure of the algorithm is similar to that without conditionals: *i.e.*, each ISG node has an associated expression enumerator, and in each iteration a new expression e_i is produced at some node N_X. State updates are a straightforward combination of Sec. 3.4.2 and Sec. 3.5.1: namely, e_i is tested for validity for both $\mathscr{E}_1$ and $\mathscr{E}_2$, for each outgo-



```
if 0 <= x:
  x = x
else:
  x = -x
```

```
if 0 <= x:
else:
  x = -x
```

```
if x < 0:
  x = -x
```

```
x, y = x + z, y + z
```

```
x = x + z
y = y + z
```

```
if 10 > a:
  a += 1
  c = a + b
  d = a + b
  b -= 1
else :
  a += 1
  d = a + b
  c = a - b
  b -= 1
```

```
a += 1
d = a + b
if 10 > a:
  c = a
else :
  c = a - b
b -= 1
```

Figure 3.9. Examples of post-processing of synthesized conditional programs to make them more readable.

ing edge $(N_X, N_{X'})_{\langle \mathscr{E}_1, \mathscr{E}_2 \rangle}$, and each variable $x \in X' \setminus X$, and the assignment $\mathscr{A}^{\mathscr{E}_j}_{(X,X')}$ is updated accordingly. Whenever a boolean expression is enumerated at the node N_{start}, the algorithm additionally tests whether it matches any partitions and updates $\mathscr{C}$ accordingly. Note that we are looking for a program with a single top-level conditional, where the condition is always evaluated in the initial state; this is why the algorithm only needs one $\mathscr{C}$, and the conditions are produced by the enumerator at N_{start}.

An edge $(N_X, N_{X'})_{\langle \mathscr{E}_1, \mathscr{E}_2 \rangle}$ is considered *complete* when both of its assignments $\mathscr{A}^{\mathscr{E}_1}_{(X,X')}$ and $\mathscr{A}^{\mathscr{E}_2}_{(X,X')}$ are complete (*i.e.*, do not contain $\perp$). The algorithm terminates when there is a partition π^* such that:

- it has a condition in $\mathscr{C}$: $\mathscr{C}(\pi^*) \neq \perp$, and
- the subgraph $\mathscr{G}/\pi$ has a path from N_{start} to N_{end} along complete edges.

The algorithm then returns the solution `if` $\mathscr{C}(\pi^*)$: q_{then} `else:` q_{else}, where q_{then} and q_{else} are sequences of assignments collected from $\mathscr{A}^{\mathscr{E}_{then}}_{(X,X')}$ and $\mathscr{A}^{\mathscr{E}_{else}}_{(X,X')}$ along the complete path (with $\langle \mathscr{E}_{then}, \mathscr{E}_{else} \rangle = \pi^*$). Once again, since multiple solutions may be discovered simultaneously, the algorithm searches for a shortest complete path in each $\mathscr{G}/\pi$, and then selects the smallest program among these candidates.

3.5.3 Post-Processing

Because the synthesis algorithm described in Sec. 3.5.2 generates programs in a restricted form (a single top-level conditional with assignments to the same variables in both branches), the resulting program is not always the most concise or natural. To improve readability of synthesized programs, LOOPY post-processes the solution to simplify it before presenting to the user.

More specifically, the following rewrite rules are repeatedly applied until a fixed point is reached:

1. **Removing self-assignment.** A variable $x \in \mathsf{Mod}_{\mathscr{E}}$ might be actually modified only in one branch of the solution and not the other. In this case, the algorithm generates a self-assignment $x = x$, which can be simply removed during post-processing, as shown in step ① in Fig. 3.9a.
2. **Removing empty branches.** As a result of applying other rules, one branch of the conditional can become empty and can be removed. If the remaining branch is the *else* branch, LOOPY turns it into the *then* branch and negates the condition, as shown in step ② in Fig. 3.9a.
3. **Splitting simultaneous assignments.** The generated solutions might include simultaneous assignments even when they are not strictly required (recall the example in Sec. 3.4.1). Our experience shows that a sequence of simple assignments is usually more familiar and hence more readable than a simultaneous assignment. For this reason, LOOPY splits a simultaneous assignment into a sequence of simple assignments whenever this is sound, *i.e.*, when there is no cross-variable dependency between left- and right-hand sides. Step ③ in Fig. 3.9b shows an example of this rewrite; this is sound because `x + z` does not use `y` and `y + z` does not use `x`. On the other hand, `x, y = x + z, y + x` cannot be straightforwardly split, because the right-hand side assigned to `y` uses `x`, and specifically its old value. Splitting assignments has the additional benefit that it makes other rewrite

rules more likely to apply.

4. **Factoring out unconditional code.** Some of the assignments might be duplicated between the *then* and *else* branches, and hence can be factored out of the conditional. An example is shown in step ④ in Fig. 3.9c. More specifically, LooPy extracts the identical prefix and suffix of assignments from the two branches, and places these statements before and after the conditional, respectively. In order to maximize the common prefix/-suffix, LooPy also re-orders assignments inside each branch whenever this is sound; for example in Fig. 3.9c the statement `d = a + b` is re-ordered before `c = a + b` (since there is no dependency between the two) and becomes part of the common prefix.

3.6 Empirical evaluation

We design our experiments to answer the following research questions: overall, we want to show that LooPy requires less time and less input from the user than big-step synthesizers, and yet generates correct and general programs most of the time.

(**RQ1**) Can LooPy handle a wide range of synthesis tasks at interactive speeds?

(**RQ2**) Does LooPy require less user input to synthesize correct programs with loops compared to the state of the art?

(**RQ3**) Does enumerating programs using the conditional ISG affect LooPy's ability to solve simple non-conditional synthesis tasks?

(**RQ4**) Are assignment sequences necessary to solve benchmarks with loops, or is simultaneous assignment sufficient?

Implementation. We implemented LooPy in Scala based on the algorithm in Sec. 3.5.2, using a standard size-based bottom-up synthesizer as the expression enumerator in each ISG node. Each enumerator has a vocabulary of 84 components, plus all the variables available in the

Table 3.1. Summary of the four benchmark suites used in LOOPY's empirical evaluation.

Benchmark suite	Number of benchmarks	Avg. $\lvert \mathrm{Mod}_{\mathscr{E}} \rvert$	Avg. solution size (AST nodes)	Avg. number of examples	Avg. iterations per example
No control flow	61	1.7	8.9	1.6	–
LOOPY conditional	9	1.0	14.6	2.6	–
LOOPY	38	1.8	11.9	1.1	4.1
FRANGEL	23	1.0	7.9	1.0	6.0

before-state, and string literals extracted from the after-state. Our implementation is single-threaded; there is much room for improvement via parallelism, as each ISG node can be handled independently.

Benchmarks. We evaluated LOOPY on 131 benchmarks from four benchmark suites:

1. *No control flow:* a suite of 61 benchmarks from previous synthesizers that generate assignments without conditions or loops.

2. LOOPY *conditional:* a suite of 9 benchmarks that contain a conditional statement outside the context of a loop.

3. LOOPY*:* a suite of 38 block-level synthesis tasks extracted from programs with loops via live execution; the original looping programs are curated from competitive programming and educational problems, as well as our user study tasks, described in Sec. 3.7.

4. FRANGEL*:* 23 benchmarks from FRANGEL's *ControlStructures* benchmark suite [161].

The statistics for these benchmarks, including the size of specification provided, are shown in Tab. 3.1. For each benchmark, we created a set of gold standard solutions to compare synthesis results against. We next detail the process of selecting and converting FRANGEL's benchmarks for LOOPY.

Selection criteria. We selected 23 tasks from FRANGEL's *ControlStructures* benchmark suite, which tests manipulating list-like data structures in various ways. Of the 40 benchmarks in the original suite, we excluded those with constructs not supported by LOOPY according to the following criteria:

– Benchmarks that contain chained conditionals or multiple chained loops. Such benchmarks were broken up into multiple benchmarks and added to the LooPy *benchmarks* set.

– Benchmarks that contain unsupported types, where no comparable type exists in LooPy (*e.g.*, matrices, benchmarks that hinge on null pointers).

– Benchmarks that require external library functions not present in the standard library.

Benchmark translation. Since FrAngel synthesizes Java programs from end-to-end specifications, we converted FrAngel's benchmarks into Python and block-level specifications. Additionally, the typical structure of a FrAngel benchmark is a set of approximately five examples where one represents the general case and the remainder are mutations covering corner cases. We converted the general case example from each of the selected FrAngel benchmarks to the LooPy specification format using the following steps:

1. The representative example was translated from Java to Python; built-in Java collections were replaced with Python lists or sets, and Java-specific elements such as `null` references were removed if present.

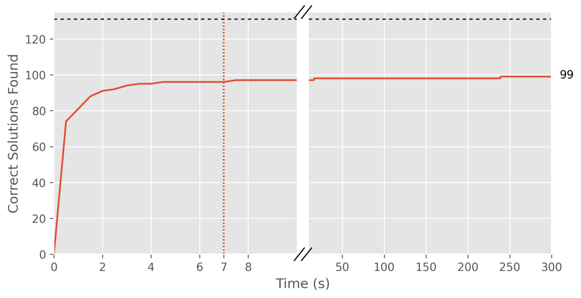
2. We manually specified the intermediate variables (if needed) to synthesize the solution.

3. We manually specified the loop structure, typically a `for` loop over the elements or the indexes of the input.

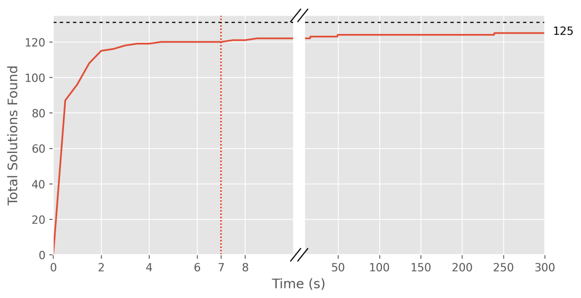
4. Starting with the representative example's input, we provided the intermediate output values for each iteration (typically six iterations).

5. FrAngel's gold standard solution was translated from Java to Python.

Additionally, benchmarks that can be solved by LooPy without the use of a loop were *also* added to the *No Control Flow* or LooPy *Conditional* benchmark suite.



(a) Correctly solved benchmarks



(b) All benchmarks that did not time out

Figure 3.10. Performance of the LOOPY synthesizer on the full benchmark suite at 300s timeout.

Experimental setup. All benchmarks were run on an AMD Ryzen 3800X processor, with the JVM maximum heap size set to 24 GB.

3.6.1 RQ1: Synthesis at Interactive Speeds

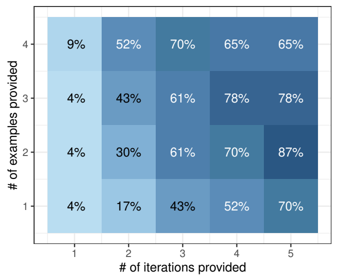
To test RQ1, we ran LOOPY on all 131 benchmarks in our joint benchmark suite. We set a timeout of five minutes, and measured the time to completion and the correctness of the synthesis result for each of the benchmarks. We show the results in Fig. 3.10.

Results. Of the 131 benchmarks, LOOPY terminates on 125 (95%) within five minutes, and correctly solves 99 (76%). However, since five minutes is far too long a wait within an actual programming workflow, we would like to examine LOOPY at an interactive timeout, seven seconds.

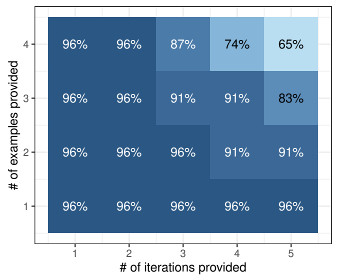
By a seven-second timeout, LOOPY terminates on 120 benchmarks (92%), it correctly solves 96 (73%). This difference is small enough to consider the shorter timeout extremely beneficial: most synthesis tasks still behave the same, but the wait is sufficiently short to prevent the user from losing the context of their work. We therefore **answer RQ1 in the affirmative**.

3.6.2 RQ2: Specifications Required for LOOPY's Interaction Model

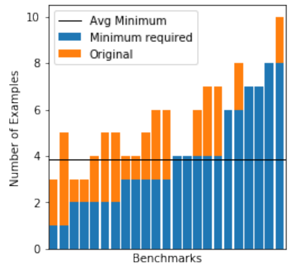
To test whether LOOPY's interaction model requires less specification effort than the state of the art, we picked FRANGEL as our baseline and used their manually crafted *Control-*



(a) Correctly solved benchmarks



(b) All benchmarks that did not time out



(c) FRANGEL end-to-end examples

Figure 3.11. The effect of the number of examples and the number of iterations from each example on LOOPY using the FRANGEL benchmark set, and the number of examples provided and required by FRANGEL.

Structures benchmarks. We performed two experiments:

1) We manually minimized the example sets in FRANGEL's original benchmarks to the minimum set required for correctness. This gives us a *lower bound* on the number of end-to-end examples that are necessary to use a big-step synthesizer.
2) We tested LOOPY with a varying number of examples and a varying number of loop iterations per example.

Minimizing FRANGEL

FRANGEL's original *ControlStructures* benchmarks have an average of 5.5 end-to-end examples per benchmark. To minimize the example sets, we modified the benchmarks by successively reducing the number of examples as long as the result remained correct. Because FRANGEL uses stochastic search, we ran each modified benchmark three times with a timeout of 30 seconds, and considered the result correct if it was correct in at least one of the three attempts.

Results. The results are shown in Fig. 3.11c. For most benchmarks, FRANGEL needs most of the original examples—an average of almost four end-to-end examples per benchmark. We further observe that even though for some of the benchmarks half or more of the examples could be removed, selecting those examples was far from intuitive. The original example set is more representative of a typical input to a big-step PBE synthesizer, since it was likely curated via the usual method of adding examples as long as the result is incorrect.

Varying number of examples for LOOPY

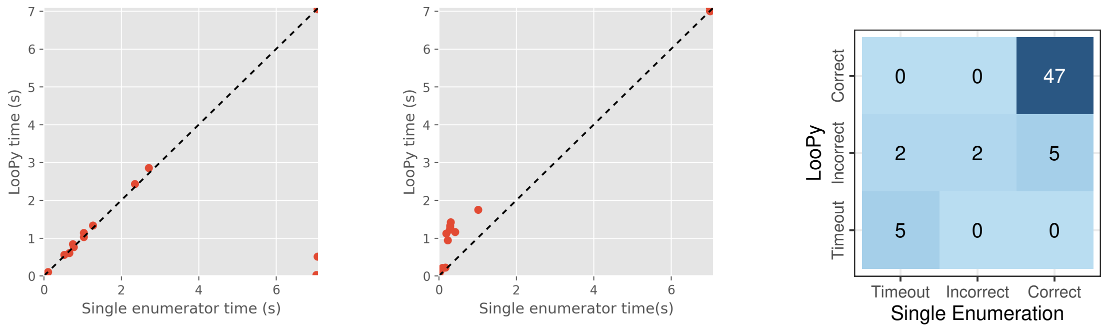
To measure the size of user input LOOPY needs, we exercised it with varying numbers of examples and loop iterations per example.

We translated each FRANGEL benchmark with all its original examples, excluding examples where the loop is not entered (*e.g.*, an empty list as input). As mentioned above, each FRANGEL benchmark file typically starts with a "main" example (sometimes two) demonstrating the main success scenario of the desired program, followed by mutations introducing corner cases. Many of these are *differentiating examples*, which do not capture representative behaviors but rather distinguish the desired program from simpler programs. For this reason, we cannot select n examples at random, as we may wind up with just the corner cases and nothing demonstrating the core behavior. Therefore, we consider the examples in the order in which they appear in the FRANGEL benchmark file.

For each FRANGEL benchmark, we run LOOPY on the first 1–4 examples, with the first 1–5 iterations of the loop for each example. The percentage of benchmarks that did not time out and the percentage of benchmarks where LOOPY found the correct program appear in Fig. 3.11.

Results. LOOPY's correctness peaks around 4–5 iterations of 2–3 examples, but even at five iterations of one example, correctness is at 70%. This is despite several FRANGEL benchmarks where the end-to-end specification requires at least two examples, such as the benchmark "Are all list elements positive?". Here, the end-to-end specification requires two examples: one where the property holds and one where it does not. In LOOPY, however, it is sufficient to invoke the program on a single list that does not satisfy the property, as long it has a prefix that does. If we use all iterations of a single example (often more than five), LOOPY succeeds on all FRANGEL benchmarks but one.

It is also rather expected that providing just one iteration has a low success rate, even



(a) Time: single assignment **(b)** Time: assignment sequence **(c)** Correctness

Figure 3.12. Differences in synthesis time and correctness between LOOPY and a single expression enumerator.

with multiple examples (see the leftmost column of Fig. 3.11a). It is similarly unsurprising that LOOPY times out more often as the number of examples increases (see Fig. 3.11b), due to the properties of Observational Equivalence we discussed in Sec. 3.4.2.

To conclude, LOOPY does quite well when provided *fewer* inputs than FRANGEL: LOOPY solves most of the benchmarks using only five inputs (more specifically, five iterations of one examples), while FRANGEL requires 5.5 examples on average. LOOPY reaches its peak accuracy with 6–10 inputs, which is comparable with the number of examples FRANGEL needs on its most demanding benchmarks. Moreover, we believe that multiple loop iterations for one example are easier to provide than to come up with entirely new end-to-end examples, which is supported by our user evaluation in Sec. 3.7. With this in mind, we **answer RQ2 in the affirmative.**

3.6.3 RQ3: The Overhead of Conditional ISG

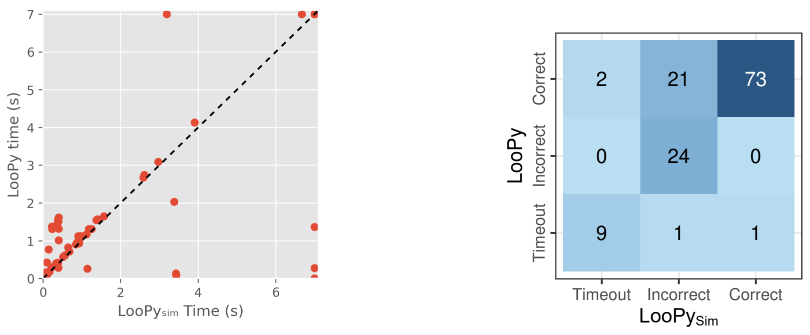
The goal of this experiment is to measure the overhead of maintaining multiple expression enumerators and keeping track of multiple example partitions. To this end, we compare the full LOOPY algorithm from Sec. 3.5.2 to a baseline synthesizer, which consists of a single bottom-up OE enumerator (of the same kind as used in each node of the ISG). Because the baseline synthesizer cannot handle conditionals or sequential composition (it can only synthesize a single assignment statement), we restrict this experiment to the *No Control Flow* benchmark suite. While none of these benchmarks require conditionals, some of them do require multiple assignments. When solving these benchmarks with the baseline synthesizer we manually decompose them into independent single-assignment tasks and set the timeout of seven seconds for each task. We then compare synthesis times and results between the baseline synthesizer and the full LOOPY algorithm; the results are shown in Fig. 3.12.

Results. We first focus on synthesis times for those benchmarks that only require a single assignment (Fig. 3.12a). We notice that with the exception of two benchmarks synthesis times remain mostly unchanged, as does the number of expressions enumerated. This is understandable because when a task has just one modified variable, the ISG contains only two nodes, N_{start} and N_{end}, and a single enumerator at N_{start} (which is identical to the enumerator of the baseline synthesizer). Hence, both algorithms are exploring exactly the same stream of expressions; the only difference in performance comes from the fact that LOOPY has to test validity of each expression against multiple partitions (and also test boolean expressions as candidates for the condition store). The two outliers are benchmarks where the baseline synthesizer times out while LOOPY finds a short but incorrect conditional solution.

For the benchmarks that require assigning multiple variables (Fig. 3.12b), LOOPY predictably takes longer, because it has to figure out the order of assignments, while for the baseline synthesizer the order is predefined. Despite this handicap, LOOPY takes no longer than two seconds to solve each benchmark that the baseline synthesizer can also solve.

There are five benchmarks that were correctly solved by the baseline synthesizer and incorrectly solved by LOOPY (Fig. 3.12c); all five require multiple assignments. In three of those, LOOPY finds a different order of assignments, which happens to match the examples; in the remaining two, LOOPY introduces a spurious condition, which leads to a shorter but incorrect solution.

Overall, using a conditional ISG has some effect on LOOPY's ability to solve bench-



(a) Running Time **(b)** Correctness

Figure 3.13. Difference in running time and correctness between LOOPY & LOOPY$_{\mathsf{sim}}$.

marks without control flow, but **this effect is small compared to the wealth of new benchmarks that can now be solved.**

3.6.4 RQ4: The Effect of Assignment Sequences

Maintaining an ISG with multiple enumerators enables LOOPY to synthesize assignment sequences, which, as we illustrated in Sec. 3.4.2, may result in a simpler solution compared to a single simultaneous assignment. In this experiment we evaluate whether this actually happens in practice (and hence, whether the additional complexity and performance overhead of the full ISG is justified). To this end, we created LOOPY$_{\mathsf{sim}}$, a version of LOOPY capable of synthesizing only simultaneous assignments but not assignment sequences. LOOPY$_{\mathsf{sim}}$ maintains a conditional ISG with only two nodes, N_{start} and N_{end}. We ran LOOPY$_{\mathsf{sim}}$ on our entire benchmark suite and measured time to termination and correctness, compared to the original LOOPY synthesizer. The results appear in Fig. 3.13.

Results. When it comes to synthesis times, on average, LOOPY tends to be slightly slower that LOOPY$_{\mathsf{sim}}$, because of the overhead of multiple enumerators. Predictably, this effect is not observed for any single-variable benchmarks, as the two ISGs are identical in this case. For most other benchmarks, the performance overhead is small: most importantly, there are only two benchmark where LOOPY$_{\mathsf{sim}}$ finishes and LOOPY times out. There are also two benchmarks

```
"""
k-th digital root:
    Compute the k-th digital root of a natural
    number n, that is, replace n with the sum
        of its digits k times.
    >>> task(1798, 3)
    7
    (because
        1. 1 + 7 + 9 + 8 -> 25
        2. 2 + 5         -> 7
        3. 7             -> 7)
"""
def task(n, k):
    rs = n
    return rs

task(1798, 3)
```

```
"""
Extract Numbers:
    Given a string containing numbers separated by
    exactly one non-integer character, return a
    list containing all the numbers in the string.
    >>> task('13a7b42')
    [13, 7, 42]
"""
def task(s):
    rs = []
    return rs

task('13a7b42')
```

Figure 3.14. The initial state of the study tasks, as provided to the users in VSCode.

where the opposite happens: LOOPY finishes and LOOPY$_{\text{sim}}$ times out[9]; this happens precisely because the simultaneous assignment solutions are larger (in this case, sufficiently large to cause a time out).

With regard to the quality of solutions, Fig. 3.13b shows that there are 21 benchmarks that LOOPY solves correctly and LOOPY$_{\text{sim}}$ solves incorrectly. Note that we did not include any programs with simultaneous assignments in the set of gold standard solutions, because we consider them less readable. As a result, we consider a LOOPY$_{\text{sim}}$ solution "incorrect" whenever it still contains a simultaneous assignment after post-postprocessing (even if it is semantically equivalent to the gold standard). Out of these 21 incorrect solutions, 10 are actually larger in size than the sequential version because they repeat sub-expressions instead of using an intermediate variable.

To conclude, with the same number of timeouts and more correct programs, we find assignment sequences to benefit LOOPY and **answer RQ4 in the affirmative**.

3.7 LOOPY in the Hands of Users

Block-level synthesis relies on user-generated block-level specifications, and we need to support the assumption that these specifications are reasonable and convenient for users to provide. To that end, we ran a preliminary qualitative user study focusing primarily on the

[9]In Fig. 3.13a there appear to be three such benchmarks, but one of them actually finishes right before the timeout.

question:

(**RQ5**) Is providing block-level specifications feasible for users?

Implementation. We modified our VSCode extension for Small-Step Live PBE [52], adding: i) sketch holes with multiple variables, ii) separation between the before- and after-state in the projection box, and iii) *live execution* of sketches with the user as oracle. A live PBE synthesis task is then converted into a block-level synthesis task as described in Sec. 3.3.

Study method. We recruited five participants (one male, one female, one non-binary, and two preferred not to state), with 4–10 years of programming experience for a two-hour study. Participants were recruited online and screened by the question "In the past year, how often did you use Python?", selecting participants who reported more than "never" and less than "once a day".

The study was conducted over a remote-controlled Zoom session on the same desktop machine used for the experiments in Sec. 4.7. In the first part of the study, users watched a tutorial video and solved a training task (String Compression from Sec. 4.2), where they could get assistance and were encouraged to ask questions. In the second part of the study, they were asked to solve two study tasks (depicted in Fig. 3.14) and explain their thought process throughout. Since the focus of the study was not on solving the tasks, but on providing specifications, users were specifically asked to solve the tasks using a loop, and if they were struggling with the algorithm after 20 minutes, we verbally provided an algorithmic hint. Each task had a 30 minute time limit. At the end of the session, users were asked to fill a short survey about their experience, what they found helpful or frustrating, and what suggestions they have for improving LOOPY.

Observations. Four users solved both tasks and the remaining one only solved the second task; two users required a hint for one of the tasks. Given the small size and scope of the study, we forgo a quantitative analysis of their sessions, and focus on qualitative responses and observations. We found that users naturally provided block-level specifications where appropriate without any notable issues in terms of the specification itself.

P1 and P3 explained they found LooPy challenging because of the need to have a concrete algorithm in mind before providing the specification. P3 stated that “the hardest part for me was the need to understand the structure of the skeleton before handing things to LooPy—how many and which variables I want, and on what to iterate”. We observed a related pattern, where P1, P2 and P4 started with a small-step specification, and once they had a more holistic view of the loop body tried again with a block-level specification for the entire loop body. P3 and P5 provided correct block-level specifications from the start. Given that P3 and P5 were not notably more experienced than other participants, we think that this is most likely because they had the complete algorithm in mind from the beginning, whereas other participants incrementally discovered the algorithm and realized after a few tries that they would need to modify multiple variables at once.

Users also found LooPy useful in multiple ways. P1 and P2 both mentioned that it was useful in writing more idiomatic code. P3 mentioned that they could have solved the tasks without LooPy, but it would have been more frustrating, and P4 found synthesizing loop bodies less tedious than writing them manually, saying “anywhere there’s a loop, and I kind of know what it’s supposed to do, [...] it’s much easier to just write the input-output examples for each loop body iteration”.

Conclusions. While we would need a much larger-scale study to make strong empirical claims, this study suggests that block-level specifications are indeed reasonable and intuitive to provide. Additionally, we notice that block-level specifications allow for a data-driven pattern of exploratory programming, where the programmer explores different intermediate states instead of code.

One major limitation of LooPy (mentioned by P1, P4 and P5) was users wanting a better understanding of why LooPy fails when it does. This is not unique to LooPy, *e.g.*, it is discussed as a part of the “User-Synthesizer Gap” in our prior work [52].

3.8 Limitations

In this section, we discuss some of the limitations to LOOPY’s generality and usability, one stemming from the interaction model and others from the UI design.

Correcting specifications. The current LOOPY UI makes it difficult for users to iterate on their specifications. While the user is providing examples in the projection box, they can move between variables and loop iterations and change their input (while live execution updates the rest of the projection box accordingly). Once they launch the synthesis task, however, the projection box disappears. If synthesis fails or produces a wrong result, the user cannot go back and edit their input; instead, they have to restart the interaction, providing the entire specification from scratch. Similarly, when the focus is inside the projection box, the user cannot modify the surrounding code or the set of output variables of the hole without exiting from the box and restarting the interaction.

The need to correct an erroneous specification has been pointed out by several of our study participants. We believe this can be fixed just by changing the UI so that undoing a synthesis task returns the user to the projection box with the latest specification. More complex forms of storing and restoring specifications, however, are non-trivial to implement, especially if the code surrounding the hole has changed, which invalidates the before-states inside the specification.

While loops. We designed and tested LOOPY in the context of `for` loops iterating over a fixed collection. In this context the number of loop iterations is known a-priori, making it natural for the user to specify one iteration at a time, as shown in Fig. 3.2. Although live execution and block-level synthesis generalize straightforwardly to `while` loops with a user-provided loop condition, the user experience is far more confusing in that case, as the number of loop iterations displayed in the projection box might change as the user is entering the specification. Specifically, assume that the user enters a hole in place of the entire body of a `while` loop. Upon entering this hole's projection box, the loop condition must evaluate to `True`, making the loop infinite; in this case the projection box displays the first several iterations. The user can then proceed to enter after-states for as many iterations as they would like. As soon as the state they entered causes the loop condition to evaluate to `False`, however, any further iterations disappear from the projection box. Next, assume that the body of the loop contains more code around the hole. Now the number of iterations may change beyond the next iteration, making the change even less comprehensible. Although this interaction is supported by LOOPY, we deemed it too confusing for the user to include in our evaluation.

Comprehensions. LOOPY inherits the ability to synthesize Python's list and dictionary comprehension from Small-Step Live PBE [52]. Unlike loops, however, the user cannot observe individual iterations within a comprehension and specify intermediate values for the data structure it is building. Instead, a comprehension is handled and specified like any other expression on the right-hand side of an assignment. This, however, is strictly a limitation of the current UI implementation: the live PBE interaction model could certainly be applied to synthesize comprehensions from block-level specifications.

3.9 Related Work

There is a long and rich history of work on program synthesis. Broadly speaking our work distinguishes itself from prior work by providing a block-level synthesis approach and associated interaction model that allows small-step synthesis with control structures at interactive speeds. We now discuss the most closely related work to LOOPY.

Synthesis with loops and recursion. Relatively few PBE tools support loops and recursion (or equivalent higher-order functions). Perhaps the most closely related to LOOPY is FRANGEL [161], which supports component-based synthesis for Java programs with control structures. Because FRANGEL uses big-step (function-level) specifications, in principle it does not require users to have knowledge of the algorithm or intermediate variables. In practice, however, to make the search tractable, FRANGEL requires users to provide a variety of examples including base cases and corner cases, and so some knowledge of the algorithm is still required. Also, as discussed more throughout the paper, FRANGEL's approach is not fast enough for an interactive setting.

Other PBE tools that efficiently support recursion and higher-order functions include ESCHER [6], MYTH [135], SMYTH [117], λ^2 [54], BIGλ [164], and RESL [141]. There is a general theme behind all these tools: efficient synthesis is achieved by extracting a local specification for the recursive call (or the higher-order argument). Different tools use different approaches to make such extraction possible. For example, MYTH requires the user examples to be *trace complete*; λ^2 does not require trace completeness, but only works efficiently when examples happen to be trace complete; RESL restricts iteration patterns to map and filter (as opposed to general folds) which enables extraction of a local specification without a trace completeness requirement. LOOPY is similar to all these tools in that it uses local specifications of loop bodies to achieve efficient synthesis, but uses a different approach to make this feasible: LOOPY supports dependent (fold-like) loops, and leverages its interaction model to solicit local specifications from the user.

Our solution to synthesizing loops by asking the user to provide more convenient specifications is partly inspired by ROUSILLON [29], a tool for web scraping by demonstration. ROUSILLON can synthesize programs with loops that extract tabular data from a webpage by making a "contract with a user" that they will demonstrate just the first row of the table; ROUSILLON even supports nested loops using the same technique and domain-specific insights. The main difference with our work is that ROUSILLON is a domain-specific end-user tool, and all of its loops are essentially maps (from the DOM to a table), whereas LOOPY handles more general loops in a context of a general-purpose programming environment.

Synthesis with conditionals. Our technique for generating conditional statements is related to the various techniques for condition abduction [104, 93, 8, 146, 161]. It is most related to the the approach used by EUSOLVER [10], which enumerates programs until a set of programs covers all input-output examples, and then attempts to synthesize a condition that separates the examples between branches; the main difference is that in EUSOLVER branches are just expressions, whereas in LOOPY branches are sequences of assignments, so testing whether all examples are covered is more involved. On the other hand, LOOPY only generates binary conditionals with an atomic condition, whereas EUSOLVER uses decision tree learning to generate multi-branch conditionals.

Synthesis with sequential composition. BRAHMA [65] proposes an efficient SMT encoding for synthesizing straight-line programs with multiple assignments to intermediate variables. Their problem is, however, very different from our assignment sequences: BRAHMA specifications are still big-step (the relation between the inputs and a single output variable), and the values of the temporary variables must be guessed by the synthesizer. Instead LOOPY takes advantage of the fact that the final values of all variables are provided to perform synthesis more efficiently using ISGs.

Synthesizers with limited support for control structures. There are also many other synthesizers in the literature, but compared to LOOPY they have limited support for control structures. This includes some interactive synthesizers that integrate into a general-purpose programming workflow, for example SNIPPY [52] and CODEHINT [57]; various other Python synthesizers, for example TFCODER [160], AUTOPANDAS [16], WREX [41]; and synthesizers for other languages [49, 72, 119, 57, 85, 196]. These all handle one-liners or sequences of method calls, with only limited support for control structures. In contrast, our proposed approach supports control structures and generating multiple statements at once.

Bottom-up enumerative synthesis. Bottom-up enumerative synthesis is a technique that originated in TRANSIT [177] and ESCHER [6], and is used in many synthesizers [160, 14, 142, 141]. This technique was originally used for enumerating expressions; we build ISGs on top of it to develop an efficient algorithm for enumerating assignments to multiple variables and introducing conditionals.

Live Execution. LOOPY's *live execution* uses concepts from *live evaluation* introduced by [132] and further adapted by [117], such as evaluating around holes and pausing the evaluation at holes that cannot be executed. Live execution is adapted from a functional domain to an imparative one, and employs no logic for resolving holes, deferring instead to an oracle—the user.

3.10 Acknowledgements

Chapter 3, in full, is a reprint of LooPy: interactive program synthesis with control structures, as it appears in the Proceedings of the ACM on Programming Languages, Volume 5, Issue OOPSLA. It was done in collaboration with Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. The dissertation author was a primary investigator and author of this paper.

Chapter 4

SOBEQ: Bottom-up Synthesis of Side-Effects with Separation Logic

4.1 Introduction

Program Synthesis is the task of automatically generating a program that satisfies a given specification. A popular specification for synthesizing general-purpose programs is input-output examples, where a solution is defined as a program that, evaluated on each provided input, produces the provided output. This is commonly referred to as *Programming-by-Example* (PBE) [63, 51, 14, 56, 71]. PBE is notable because, unlike logic-based specifications [147, 167, 84], it does not require any specialized knowledge to use, and has thus been used to develop synthesizers for end-users [63, 185] and novice programmers [52, 51].

For example, consider a synthesis specification for a JavaScript program where, given an array of integers `arr` of length `n`, we want to return the sum of the smallest and largest elements of the array. We might specify this problem with the example:

$$\{\texttt{arr} \mapsto [10, 100, 90, -1, 2], \texttt{n} \mapsto 5\} \to 99$$

If we assume the value in `arr` is disposable, e.g., it can be sorted in place, the following is a solution[1] that we would like to synthesize:

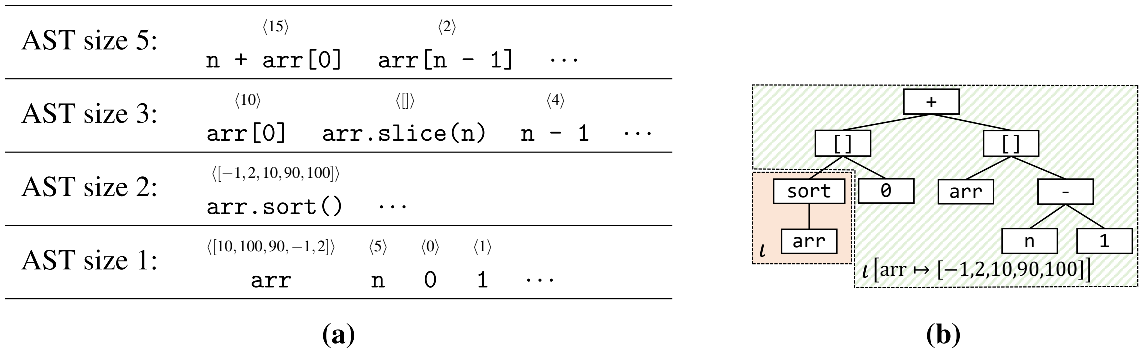


Figure 4.1. (a) Bank of programs with their observed evaluation results used by classical OE to enumerate pure programs. Size 2 contains `arr.sort()` that sorts in-place and makes compositions with it incorrect. (b) AST of the target program, divided into the concrete states in which each subexpression is evaluated.

```
arr.sort()[0] + arr[n-1]
```

To solve this synthesis task, we can specify a set of *components* (operations, library functions, etc.) and search the space of all programs that use those components. There are many of ways to perform such a search, including constructing a representation of the space that can be traversed [49, 119, 63] and encoding the problem for a solver [167, 88]. Such approaches place heavy constraints on which components can be included. But the simplest form of search is to enumerate the space of programs [47, 54, 70, 161] by applying the components, then test resulting programs. Because the space of programs is astronomical, this approach is prohibitively slow when implemented naively. Moreover, most mechanisms to prune the space are domain-specific and restricted by the manual effort for handling each component individually.

Bottom-up Enumeration. One technique that does not require any domain-specific effort is *bottom-up enumeration* with an *Observational Equivalence* (OE) reduction [177, 6]. Programs are enumerated using the components, combining smaller programs into increasingly larger ones. To prune its search space, it enumerates by constructing a bank of programs that are representative of *equivalence classes* of programs that are observationally equivalent, i.e., pro-

[1]This program should use `arr.sort((n1,n2) => n1 - n2)`, otherwise JavaScript sorts the string representations of elements lexicographically. We elide the argument for brevity.

grams that, given the set of available inputs, evaluate to the same output value.

Each new program is obtained by applying a component from the component set to smaller sub-expressions already in the bank. For example, in Fig. 4.1a applying `+` to the representative of equivalence classes $\langle 5 \rangle$ and $\langle 10 \rangle$ will yield the program `n + arr[0]`, which will be banked as the representative of equivalence class $\langle 15 \rangle$. This bank allows the enumeration to memoize the evaluation results of each program, and look them up when evaluating larger programs, rather than re-evaluating each program from scratch. OE reduces the space by keeping only one program per equivalence class. This vastly reduces the search space while ensuring no solution that existed in the un-reduced space is lost. Previous works use variations of this technique in synthesizers for C [177], Python [52, 51], JavaScript [141], Java [57], Selenium [108], and OCaml [6, 127].

Enumeration with Side-Effects. However, a common limitation of these works is handling *side-effects*. Notice that JavaScript's `sort()` works in-place. In the example above, the call to `arr[n-1]` evaluates to a different value if evaluated before `arr.sort()` versus after it. A classical OE enumerator that performs its OE-reduction based on variable values in the inputs would consider `arr[n-1]` to be in the equivalence class $\langle 2 \rangle$. However, unlike its equivalent in languages like Python, `arr.sort()` also returns a self-reference, allowing the result to be composed into larger expressions, as in our target program. If the enumeration itself does not account for this, it will use the wrong memoized expression computed using the unsorted value of `arr` in its candidates, yielding a result that does not actually satisfy the specifications. Moreover, the enumeration may see another program whose value is *really* $\langle 2 \rangle$, and discard `arr[n-1]` and never find this solution.

In other words, the classical bottom-up enumeration with OE described above loses its correctness in the presence of side-effects. Previous works on bottom-up enumeration either explicitly assume that all components are *pure* and cause no side-effects [127, 52, 185, 177, 6, 108], make assumptions that greatly reduce the space of intermediate states [51, 145], or ignore the issue, leading to an incorrect enumeration [57, 141].

Our appraoch. We introduce SOBEQ (*Side-effects in OBservational EQuivalence*), to our knowledge the first bottom-up proof-directed synthesis technique. SOBEQ is a bottom-up enumerative algorithm proven correct in the presence of mutating components.

SOBEQ is based on Concrete Heap Separation Logic (CHSL), which is a representation of the operational semantics of the language in the style of Separation Logic [151, 137]. For example, in CHSL we would describe the mutating program `n++` evaluated on the example's input as the triple $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ where the precondition $\texttt{n} \mapsto 5$ describes `n`'s initial value, and the postcondition $\texttt{n} \mapsto 6; 5$ describes its value after mutation and its result. Instead of searching for a program, SOBEQ searches for its CHSL derivation. This gives SOBEQ two key advantages.

First, SOBEQ's bank stores CHSL triples rather than programs. To build larger programs, SOBEQ also applies each component to smaller triples in the bank, but unlike simple programs, triples can only be combined according to the rules of the logic. This means programs always compose in a way that forms a valid evaluation, and there is no risk of evaluating a program in the wrong state. Moreover, OE can be determined by simply ignoring the program and comparing the specifications surrounding each triple: two programs with the same pre- and postcondition are guaranteed to behave the same, and so will also compose the same. This maintains SOBEQ's *correctness* in the presence of side-effects.

Second, CHSL can reason about the local effects of programs: they do not need to mention parts of the state that the program does not touch. For instance, $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ does not specify anything about `arr`, because `n++` will behave the same way in all concrete states where the value of `n` is 5. This representation is *compact*, with each triple describing the behavior in many, potentially infinitely many, concrete states. Moreover, we can find more general triples, ones that provide the same result and the same mutations for more concrete states. Discarding the less general programs lets us compact the space even further.

We implemented SOBEQ as a synthesizer for JavaScript programs that can create non-pure expressions and sequences of expression statements. We evaluated our synthesizer on 46 benchmarks curated from the literature, from STACKOVERFLOW and from competitive programming website LEETCODE. SOBEQ's solutions are, overwhelmingly, not overfitted and do not contain spurious mutations. Moreover, SOBEQ's runtime and number of benchmarks solved is comparable with state of the art synthesis of general mutations, but unlike previous state of the art, SOBEQ is both deterministic and proved correct.

The main contributions of this paper are as follows:

▷ A formulation of the PBE problem with side-effects as the search for a derivation in a restricted Separation Logic, CHSL.

▷ The SOBEQ algorithm, a bottom-up enumerative algorithm that searches for a derivation with an OE-reduction.

▷ An implementation of SOBEQ in a JavaScript synthesizer, evaluated on 46 benchmarks curated from the literature, STACKOVERFLOW and LEETCODE.

4.2 Overview

In this section, we present SOBEQ through the example introduced in Sec. 6.1. First, we present SOBEQ's notation for mutating programs, which is compact compared to the real space of programs. Next, we show how this helps us enumerate programs correctly. Finally, we show how ordering programs by generality helps us prune the space even further.

4.2.1 Mutating programs

We begin with a reminder of programs and their construction in Observational Equivalence [141, 6, 177], then show how we modify them to correctly reason about side-effects.

Classical OE-reduction Program Bank. Bottom-up enumeration with an OE-reduction uses a set of components, $\mathscr{C}$, to solve a synthesis task specified by a vector of input-output examples, $\mathscr{E}$, where each example is of the form $\iota \to \omega$. For each enumerated program, we then compute

$\texttt{arr} \mapsto [-1,2,10,90,100] * \texttt{n} \mapsto 5$:	$\overset{\{\texttt{arr} \mapsto [-1,2,10,90,100] * \texttt{n} \mapsto 5; 100\}}{\texttt{arr[n-1]}}$	$\cdots$		
$\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5$:	$\overset{\{\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5; 2\}}{\texttt{arr[n-1]}}$	$\overset{\{\texttt{arr} \mapsto [10,100,90,-1,2] * \texttt{n} \mapsto 5; -1\}}{\texttt{arr[n-(1 + 1)]}}$	$\cdots$	
$\texttt{arr} \mapsto [-1,2,10,90,100]$:	$\overset{\{\texttt{arr} \mapsto [-1,2,10,90,100]; [-1,2,10,90,100]\}}{\texttt{arr}}$	$\overset{\{\texttt{arr} \mapsto [-1,2,10,90,100]; -1\}}{\texttt{arr[0]}}$	$\cdots$	
$\texttt{arr} \mapsto [10,100,90,-1,2]$:	$\overset{\{\texttt{arr} \mapsto [10,100,90,-1,2]; [10,100,90,-1,2]\}}{\texttt{arr}}$	$\overset{\{\texttt{arr} \mapsto [10,100,90,-1,2]; 10\}}{\texttt{arr[0]}}$		
	$\overset{\{\texttt{arr} \mapsto [-1,2,10,90,100]; [-1,2,10,90,100]\}}{\texttt{arr.sort()}}$	$\overset{\{\texttt{arr} \mapsto [-1,2,10,90,100]; -1\}}{\texttt{arr.sort()[0]}}$	$\cdots$	
$\texttt{n} \mapsto 5$:	$\overset{\{\texttt{n} \mapsto 5; 5\}}{\texttt{n}}$	$\overset{\{\texttt{n} \mapsto 5; 4\}}{\texttt{n - 1}}$	$\overset{\{\texttt{n} \mapsto 6; 5\}}{\texttt{n++}}$	$\cdots$
emp:	$\overset{\{\mathit{emp}; 0\}}{\texttt{0}}$	$\overset{\{\mathit{emp}; 1\}}{\texttt{1}}$	$\overset{\{\mathit{emp}; -1\}}{\texttt{-1}}$	$\cdots$

Figure 4.2. The precondition bank used to enumerate the program in Fig. 4.1b. Each program is additionally labeled with its postcondition comprising an assertion and its result.

the label of its equivalence class. In classical OE, this label is $[\![p]\!](\iota)$, or its evaluation result, for each ι in the examples. In our example, `arr` has the *observed behavior* $[10,100,90,-1,2]$ on the provided input, so its equivalence class is labeled $\langle[10,100,90,-1,2]\rangle$.

To enumerate the space of programs, we use a *bank* to store previously seen programs, as seen in Fig. 4.1a: each program in the bank represents an equivalence class of programs, according to its label. For each component $t \in \mathscr{C}$ with arity k, we collect all k-tuples of programs already in the bank and use them as arguments to apply t. This is generally done with some notion of iterations, e.g., enumerating programs by increasing AST height, or, as in Fig. 4.1a, number of AST nodes. For instance, when enumerating programs of AST size 5, the enumerator considers the component array dereference, which has an arity of 2. Among pairs of programs collected from the bank with a total size of 4 will be the program `arr`, representative of the equivalence class $\langle[10,100,90,-1,2]\rangle$, and the program `n - 1`, representative of $\langle 4\rangle$. Applying array dereference will yield `arr[n - 1]` with the observed behavior $\langle 2\rangle$. Crucially, we do not have to evaluate the full program `arr[n - 1]` to get this value: we can use the observed values $[10,100,90,-1,2]$ and 4 and only compute the final step.

When a program is enumerated from an equivalence class that has no representative in the bank—i.e., no program in the bank has the same label—it is added to the bank to be used when constructing larger programs. This is the case with `arr[n - 1]`, which is added to the bank at size 5. On the other hand, if another representative of the equivalence class already exists in the bank, we simply discard the program. If a program's observed values are equal to the provided outputs, the program is a solution. Since we only use discovered equivalence classes (i.e., programs) to construct larger programs, the effect of this pruning technique compounds.

The problems with side-effects. We quickly run into issues when $\mathscr{C}$ contains components with side-effects. The next step in solving our example is to enumerate `arr.sort()[0] + arr[n - 1]`. But, as we are about to see, using the classical approach this program will not be evaluated correctly, and it may even be pruned away.

We already saw how the subprogram $p_1 =$ `arr[n - 1]` was enumerated and labeled $\langle 2 \rangle$, and $p_2 =$ `arr.sort()[0]` will be enumerated similarly: at size 4, it will be enumerated using `arr.sort()` (size 2) and `0` (size 1), and added to the bank as a representative of $\langle -1 \rangle$.

The source of our problem is that OE is a dynamic programming algorithm: when constructing the larger p_2 `+` p_1, the enumerator does not evaluate the full program. Instead, it uses the memoized observed values and adds: $\langle -1+2 \rangle = \langle 1 \rangle$. However, since p_2 modifies `arr`, p_2 `+` p_1 *actually* evaluates to 99 on the input ι. Worse, this erroneous value means the synthesizer will consider p_2 `+` p_1, our solution, equivalent to p_1 `+` p_2, as well as to `1`, so it will certainly be discarded. While a seemingly simple solution is to just evaluate the program in full each time, ensuring labels are correct, this might be expensive depending on the operation, and more importantly, would not be sufficient. If the synthesizer sees the constant value 2, e.g., by enumerating `1 + 1`, this would cause the larger—and seemingly equivalent—`arr[n - 1]` to be discarded. OE hinges on the enumeration only discarding a program when it already has a suitable replacement, but `arr[n - 1]` evaluated *after* `arr.sort()` has no equivalent program in the bank.

Rethinking the program representation. The problem above is caused by not distinguishing between the *same* program when evaluated using *different* values of `arr`. Likewise, we must also consider programs that return the same value, but their *effect* on variables is different: e.g., both `n` and `n++` evaluate to 5, but `n++` changes `n` whereas `n` does not.

Our first step, then, is to include this information alongside the program: we represent each program in our space as a triple in Concrete Heap Separation Logic (CHSL), a flavor of Separation Logic [151, 137] representing only concrete values: the precondition records the variables before its evaluation, and the postcondition comprises two elements, an assertion recording the variables after evaluation and the (concrete) result of the evaluation, like so:

$$\{\texttt{arr} \mapsto [10,100,90,-1,2]\}\texttt{arr.sort()}\{\texttt{arr} \mapsto [-1,2,10,90,100];[-1,2,10,90,100]\}$$

where $\texttt{arr} \mapsto [-1,2,10,90,100]$ denotes the local heap after `arr` is sorted in-place, and $[-1,2,10,90,100]$ denotes the self-reference returned by `sort()`. For the remainder of the section we will use arr_{orig} for the original array value and arr_{sort} for its sorted value, for brevity.

Notice that because all values are concrete, we can still rely entirely on evaluation rather than on domain-specific logical inferences. CHSL is defined in Sec. 4.3.1.

In this representation, we can tell apart $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{orig}; arr_{orig}\}$ and $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$, or `arr` evaluated on the original and the sorted array. Once `arr` is modified by `sort()`, we need the second one.

Notice that these pre- and postconditions do not encode fully-concrete states; this makes our representation of the space more compact. The triple $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ leaves `n` unconstrained, taking advantage of the fact that only constraining `arr` is sufficient to represent all variable valuations relevant for this program. In other words, we only need to constrain the *footprint* of the program.

CHSL includes the notion of the separating conjunction $*$ to indicate separate parts of the heap (or, in our case, variable store) that can be reasoned about locally. Thanks to aliasing restrictions of the synthesizer's target language detailed in Sec. 4.3.1, any two variables can be separated with $*$. E.g., the assertion $\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5$ describes the concrete initial state. With this we can now further borrow from Separation Logic when considering evaluation sequences.

4.2.2 Enumeration and heaps

A bottom-up enumeration composes larger programs from smaller ones, or, in other words, selects a sequence of arguments to apply component t to. To evaluate such a composition the arguments to t are first evaluated in sequence, and their results are then passed to t for evaluation. With a mutation-free component set, any sequence of arguments is fine: since all programs begin at a precondition that describes the initial state and end at a postcondition that still describes the initial state, evaluating any sequence of programs is correct. However, let us consider two programs in our space, `arr[0]` and `arr.sort()[0]` enumerated with classical OE in Fig. 4.1a.

The first is now the triple $tr_1 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{orig}; 10\}$ and the second is $tr_2 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$. If we consider the application of `+` to two possible argument pairs, (tr_1, tr_2) and (tr_2, tr_1) we can more easily see that for the composition to behave as expected, i.e., to apply CHSL's EVAL rule (defined in Fig. 4.4), the postcondition of each argument must be equal to the precondition of the next one in the sequence; this means all mutations are accounted for.

This means we *cannot* compose (tr_2, tr_1) under any binary operator, but the enumerator can—and will—select (tr_1, tr_2) as arguments to `+`. All that remains, then, is for EVAL to use the language interpreter to apply `+` and add the two precomputed results $10 + -1$, and get: $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0] + arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; 9\}$ whose precondition is the precondition of tr_1, and, as `+` has no additional effect, whose postcondition contains the compounded effects of the arguments.

If we partition our bank of programs by the program's precondition, as in Fig. 4.2, then the enumerator can create correct sequences of arguments by construction, starting with some triple as the first argument, then fetching all suitable next children according to their precondition.

Discovering more preconditions. Let us consider how we enumerate the last subprogram of our target program, `arr[n - 1]`. The enumerator applies array dereference to two arguments. The first is $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$, or `arr` evaluated on the already-sorted array. Importantly, this is a program that will not be enumerated if the enumerator only looks for programs that start at preconditions describing the initial state. However, it is necessary for the bottom-up construction of the target program.

To this end, whenever a new assertion is discovered by the enumerator, e.g., when a mutating component is evaluated, creating a never-before-seen assertion in the postcondition, as enumerating $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ does, this assertion is added to the precondition bank. Along with this assertion, the enumerator also adds each variable in the newly-discovered assertion with its newly discovered value to the bank as well. In our case, the triple $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ that we need is also added to the bank.

Programs with different footprints. Now that we have the first argument to the array dereference, let us consider the second. $\{\texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{n} \mapsto 5; 4\}$ is enumerated at the initial value of `n`, i.e., without the need for its precondition to be discovered. However, when the enumerator tries to compose both into a sequence to provide them as arguments to `[]`, it runs into a problem: the EVAL rule does not apply here. The assertion in the postcondition of the first argument, $\texttt{arr} \mapsto arr_{sort}$, is different than the precondition of the following argument, $\texttt{n} \mapsto 5$.

However, we clearly see that these two assertions deal with separate parts of the heap, and do not interfere with each other. This means we can apply the FRAME rule: if we add the constraint $\texttt{n} \mapsto 5$ to both the precondition and postcondition of the first triple, and likewise add $\texttt{arr} \mapsto arr_{sort}$ to both pre- and postcondition of the second triple, the more constrained triples are now composable. Then the enumerator can then apply array dereference to them, yielding $\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr[n - 1]}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; 100\}$. The full derivation is shown

$$\textsc{Frame}\ \dfrac{\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}}{\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; arr_{sort}\}}$$

$$\textsc{Eval}\ \dfrac{\downarrow \qquad \textsc{Frame}\ \dfrac{\textsc{Eval}\ \dfrac{\{\texttt{n} \mapsto 5\}\texttt{n}\{\texttt{n} \mapsto 5; 5\} \qquad \textsc{Frame}\ \dfrac{\{emp\}\texttt{1}\{emp; 1\}}{\{\texttt{n} \mapsto 5\}\texttt{1}\{\texttt{n} \mapsto 5; 1\}}}{\{\texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{n} \mapsto 5; 4\}}}{\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; 4\}}}{\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr[n - 1]}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; 100\}}$$

Figure 4.3. Using FRAME and EVAL to enumerate `arr[n - 1]`. Recall arr_{sort} is the constant value $[-1, 2, 10, 90, 100]$.

in Fig. 4.3. Then similarly, the same extension—this time, adding $\texttt{n} \mapsto 5$ to tr_2 argument and nothing to the second—is applied to allow the enumerator to compose the target program.

We notice that in this case, this will discover a new assertion not as a postcondition as we did when `sort()` mutated `arr` but as a precondition, and if this were not the target program, the enumerator would continue to enumerate more programs with this new assertion as their precondition.

4.2.3 Equivalence and mutation

So far, we showed how the enumerator composes programs, but we did not show how the OE-reduction works. Now that we have a new representation of programs, we can re-define how each triple is observed and labeled, so that the label vector suits our needs.

In classical OE, the observed value for each example is the program's evaluation result, but this is now insufficient: while in the precondition $\texttt{arr} \mapsto [10, 100, 90, -1, 2]$ the program `arr[1]` evaluates to 100, if we discard all the following programs that evaluate to 100 we will also discard the `arr[n - 1]` that is evaluated after the array is sorted. Likewise, `n` and `n++` with the same precondition are distinct in that they return the same value given the same precondition, but have a different effect on the state.

Generally, we can see that we can label a triple with its pre- and postcondition, including the result. Once those are identical, any programs that form a valid triple with them would be *interchangeable*, i.e., can be swapped for each other with no change to a larger evaluation. In Fig. 4.2, the precondition component is expressed by the precondition the triple is stored under in the bank, while the postcondition appears above each program.

Generality of equivalence classes. The equivalence class labels above define an *equivalence relation* for OE. However, in this setting, we can do even better: there are cases where we want to discard programs even though one is not equivalent to the other. Let us consider two triples, $tr_3 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.length}\{\texttt{arr} \mapsto arr_{orig}; 5\}$ and $tr_4 = \{emp\}\texttt{5}\{emp; 5\}$, where emp is the unconstrained heap.

Both triples have the same evaluation result (5), and the same effect (no effect), but different assertions at the pre- and postconditions. But they are not completely different: if we select a *frame axiom* $R = \texttt{arr} \mapsto arr_{orig}$, we can bridge their difference: $\{emp * R\}\texttt{5}\{emp * R; 5\}$ is equivalent to tr_3. Under this condition, we say that tr_4 is more general than tr_3: it describes a program that behaves the same over more concrete states. If we let the more general equivalence class *subsume* less general ones, we arrive at a more compact (but still correct) representation of the space. While we cannot tractably apply this additional reduction to the entire space, we can at least discard less general programs once more general ones exist in the bank.

Temporary values and variables. Once we employ this tactic, will tr_4 also subsume $tr_5 = \{\texttt{n} \mapsto 5\}\texttt{n}\{\texttt{n} \mapsto 5; 5\}$ the same way it subsumed tr_3? Likewise, in Fig. 4.2, why does $\{emp\}$ `-1`$\{emp; -1\}$ not subsume $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$ and $\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5\}$ `arr[n - (1+1)]`$\{\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5; -1\}$, which are still in the bank?

If we let this happen, the programs will no longer be interchangeable, and some will break entirely, e.g., the application of `++`. While $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6; 5\}$ uses tr_5 as its argument, `5++` does not compile. Likewise, replacing `arr` within `arr.sort()` with the array literal with `arr`'s value would produce a different effect, indicting they should be separate programs.

To solve this, we separate programs returning a temporary value from those whose value is accessible from the program variables. We do this by using an evaluation result comprising two components: the *value* returned by the evaluation, and the *location* of that value, where all temporary values are given the same location $\perp$ to unify them. This way, tr_3 and tr_4 both return $5@\perp$, but tr_5 returns $5@n$. Once the definition of the postcondition deals in results rather than values, both equivalence and subsumption are correct and will differentiate between the two. Fig. 4.2 omits this for simplicity, but does still separate $\{emp\}$ `-1`$\{emp; -1\}$ with result $-1@\perp$ from $\{$`arr` $\mapsto arr_{sort}\}$`arr[0]`$\{$`arr` $\mapsto arr_{sort}; -1\}$ with result $-1@arr[0]$ and from $\{$`arr` $\mapsto arr_{orig} *$ `n` $\mapsto 5\}$`arr[n-(1+1)]`$\{$`arr` $\mapsto arr_{orig} *$ `n` $\mapsto 5; -1\}$ with result $-1@arr[3]$.

Defining the Solution. We are almost finished: like classical OE, we continue iteratively enumerating programs into the bank until a solution is found. Our final step is to reconsider the definition of a solution to a synthesis specification.

When enumerating a program in a non-mutating space with an OE-reduction using the examples' inputs, a solution is a program that is labeled by (i.e., that evaluates to) the provided outputs. In SOBEQ this is insufficient: the enumerator can find a program returning the correct value in *some* precondition. To be correct when evaluated on the concrete example inputs, we require a solution's precondition describe the concrete initial state ι. In our example, `arr` $\mapsto arr_{orig} *$ `n` $\mapsto 5$ trivially describes ι—it is equal to it—but so does `n` $\mapsto 5$, for example.

Moreover, our example started with the assumption that the values `arr` are disposable, but this may not be the case. We therefore allow the user to constrain the effects of the program, e.g., prohibiting `arr` from mutating, forcing a solution where `arr` is copied, or requiring that `arr` be modified to a new value of $[100, 90, 10, 2, 1]$, which means the maximum value will need to be obtained with `arr.reverse()[0]`. These would constrain the postcondition of a solution as well.

Thus, to be a solution, a triple must: 1) return the values of the example outputs, 2) have a precondition that describes ι, and 3) satisfy any provided constraint on the postcondition. SOBEQ is, to our knowledge, the first proof-directed synthesis algorithm that works bottom-up. It is also the first bottom-up enumerative synthesizer for general mutating components that has

Input variables	Identifiers $x \in \mathsf{Vars}$
Literal values	$v \in \mathscr{K}$
Assertion	$a \in \mathscr{A} ::= emp \mid x \mapsto v * \mathscr{A}$
Assertions	$\vec{a}, \vec{P}, \vec{Q} \in \mathscr{A} \times \cdots \times \mathscr{A}$
Location	$l \in \mathscr{M} ::= \bot \mid v \mid v[n]$, n is an integer literal
Result(s)	$r \in \mathscr{R} ::= v@l$, $\vec{r} \in \mathscr{R} \times \cdots \times \mathscr{R}$
Postcondition	$\{\vec{Q}; \vec{r}\}$

$$\frac{\{\vec{P}_1\}c_1\{\vec{P}_2;\vec{r}_1\} \quad \{\vec{P}_2\}c_2\{\vec{P}_3;\vec{r}_2\} \quad \cdots \quad \{\vec{P}_k\}c_k\{\vec{P}_{k+1};\vec{r}_k\} \\ t \in \mathscr{C}, arity(t) = k \qquad \forall i.(t(r_1^i, \ldots, r_k^i), P_{k+1}^i) \rightarrow (r^i, Q^i)}{\{\vec{P}_1\}t(c_1, \ldots, c_k)\{\vec{Q};\vec{r}\}} \textsc{Eval}$$

$$\frac{\{\vec{P}\}c\{\vec{Q};\vec{r}\}}{\{\vec{P} * \vec{R}\}c\{\vec{Q} * \vec{R};\vec{r}\}} \textsc{Frame}$$

Figure 4.4. CHSL syntax and inference rules

a correctness guarantee: if a solution exists in the space spanned by $\mathscr{C}$, SOBEQ will find an equivalent solution.

4.3 Enumerative Synthesis and Side-Effects

In this section we will define the basic elements with which we construct SOBEQ: programs and their evaluation, and the synthesis task.

We therefore begin with the building blocks for our revised definition of the synthesis task, followed by the definition of the program space and observational equivalence in that space.

4.3.1 Concrete Heap Separation Logic

In this section, we introduce Concrete Heap Separation Logic (CHSL), a notation for operational semantics in the style of Separation Logic.

The heap. In CHSL, we use a heap notation to reason locally about the concrete values assigned to variables from the input. We can reason locally about each individual variable because SOBEQ's target language does not allow assignments, and so as long as nothing is aliased at the input—we enforce this when generating the synthesis task—each variable is separate from all others and can be separated from them via $*$.

CHSL assertions. CHSL reasons about concrete values assigned to variables. As such, a CHSL assertion comprises heaplets of the form $x \mapsto v$ where v is a concrete value.

This also means we can trivially transform a state (partial function) σ into an assertion describing an identical heap s.t. $x_1 \mapsto \sigma(x_1) * \cdots * x_n \mapsto \sigma(x_n)$ for all $x_i \in dom(\sigma)$. We often use the partial function for a variable valuation interchangeably with its assertion equivalent, i.e., use σ as a shorthand for the assertion above.

To describe a vector of states $\vec{\sigma}$, corresponding to the vector of examples $\mathscr{E}$, CHSL likewise uses a vector of assertions. We therefore also define a vector separating conjunction s.t. $\vec{P} * \vec{R} = \langle P_i * R_i \rangle_i$.

Values with location. In the presence of side-effects, we must also consider that some values are accessible from our heap. As shown in Sec. 4.2, this is important for the semantics of components that modify their arguments (e.g. `arr.sort()`), and more importantly for components that return a self-reference and are therefore composable, as in `arr.sort().reverse()`, where both `sort` and `reverse` mutate `arr`.

To support this, a result $r \in \mathscr{R}$ comprises two components: a *value* and that value's *location*, $l \in \mathscr{M}$ that can be a variable, a variable at a *concrete* index, or $\perp$. We use $\perp$ to indicate a value inaccessible via any of the variables asserted over, i.e., a temporary value. We decompose the result as $r = v@l$.

Triples. A triple in CHSL comprises a precondition $\vec{P} \in \mathscr{A}^k$ (where $|\mathscr{E}| = k$), the program p, and a postcondition. Our postconditions have two components: i) an assertion $\vec{Q} \in \mathscr{A}^k$, and ii) the evaluation results of p on $\vec{P}$. As with variable values, the values in results are concrete. Notice that vectors of assertions and results form a conjunction: $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ means $\{P_1\}p\{Q_1;r_1\}$, $\{P_2\}p\{Q_2;r_2\}$, and so on until $\{P_k\}p\{Q_k;r_k\}$ all hold.

We consider $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ to be a *valid* triple if it accurately tracks the evaluation of p on $\vec{P}$. In other words, assuming a (deterministic) small-step operational semantics for programs the behavior of which is defined by the interpreter of the language: $\to$: $(\mathbb{P}, \mathscr{A}) \to (\mathscr{R}, \mathscr{A})$, we say $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ is valid if $\forall 1 \leq i \leq k.(p, P_i) \to (r_i, Q_i)$.

Because SOBEQ will be *enumerating* triples, it is crucial that it can construct valid triples.

The* EVAL *rule. When constructing a new triple, the EVAL rule denotes the use of the interpreter to ensure the resulting triple is valid. Because an evaluation of an AST node will first evaluate its children in sequence, then use the results to evaluate the node's operation, EVAL requires a sequence of triples. As in the classical Hoare Logic SEQUENCE rule, EVAL requires matching midconditions. However, since we often want to form a sequence from triples where $Q_i \neq P_{i+1}$, we use the FRAME rule to find a larger composed heap where the midconditions do match.

The* FRAME *rule. Generally, all triples in the space contain only their footprint in their pre- and postcondition. However, the EVAL rule requires a sequence of arguments with matching midconditions, and, as the example in Fig. 4.3 shows, in this compact representation this is not always the case.

We recall that the frame rule allows us to reason about a larger heap by equally extending the pre- and postcondition with an additional assertion called the *frame axiom*.

Fortunately, in CHSL, frame axioms are very easy to find. To prepare a pair of arguments for EVAL, then, one of two things is true: either for every i, $Q_1^i \cup P_2^i$ is still a partial function (i.e., if the same variable appears in both, it maps to the same value) and R_1^i, R_2^i are found by their difference, or the two triples cannot form a valid sequence.

4.3.2 Problem definition

Since SOBEQ uses CHSL triples to solve a PBE problem, we must re-state the PBE problem for CHSL. In this section we define what it means for a CHSL triple to represent our target program, define a translation from a PBE task, and explain how additional specifications, constraining the program's effects on the state, can also be supported.

Usually, the Programming by Example (PBE) task is defined as follows: given a vector of input-output examples $\mathscr{E} = \langle \iota_i \to \omega_i \rangle_i$, find a program p over a set of components $\mathscr{C}$ where for every input ι_i, evaluating the program on ι_i, denoted $[\![p]\!](\iota_i)$, is equal to the provided output ω_i. We first want to translate this into a goal of the shape $\{P\}\_\{Q;r\}$.

Examples in CHSL. We denote the vector of input states $\vec{I} = \langle \iota_i \mid \iota_i \to \omega_i \rangle_i$. The direct translation of $\mathscr{E}$ to CHSL, then, is $\{\vec{I}\}\_\{\_;\langle \omega_i @\_ \rangle_i\}$. In other words, any postcondition will satisfy our specification, and the correct values can be at any location.

Constraining effect. We notice that the user may want to constrain specific mutations as well as the value. We let the programmer attach to each example any constraints on the effect, e.g., the value of `n` must be 9, or `arr` must end in its initial value $[10, 100, 90, -1, 2]$. To do this, we let the user provide an *effect constraint* $q : \mathsf{Vars} \to \mathscr{K}$, the constrained target value for any variable, where if $x \notin dom(q)$, x is unconstrained.

The* SOBEQ *task. The user, then, does not need to know about CHSL. Much like the original $\mathscr{E}$, they provide a vector $\vec{\Phi}$ such that each $\varphi \in \vec{\Phi}$ is a pair of example and effect constraint: $\varphi = (\iota \to \omega, q)$. If they are unconcerned with effects, this is identical to providing $\mathscr{E}$.

We then turn $\vec{\Phi}$ into a synthesis goal in CHSL:

Definition 5. (SOBEQ synthesis goal) Given a specification $\vec{\Phi}$, our goal $\mathscr{G}_{\vec{\Phi}}$ is:

$$\mathscr{G}_{\vec{\Phi}} = \{\vec{I}\}_{\_}\{\langle q_i * \_\rangle_i ; \langle \omega_i @ \_\rangle_i\}$$

and the SOBEQ task is to find a triple $\{\vec{\sigma}\}p\{\vec{\sigma}'; \vec{r}\}$ that matches $\mathscr{G}_{\vec{\Phi}}$.

4.3.3 The program space

Now that we have SOBEQ's specifications, we define the space in which it performs the search. In the remainder of this section, we recall the definition of classical OE, define the space in the presence of mutations, and partition that space into equivalence classes.

The full program space of a synthesizer with a component set $\mathscr{C}$ is every program using the components in $\mathscr{C}$ (including the variables in $\vec{I}$). We denote this set $(\!|\mathscr{C}|\!)$. This set is infinite, but more importantly highly redundant, e.g., contains both `x + 1` and `x + 0 + 1`, and more relevant to our mutating domain, `arr.sort()` and `arr.sort().sort()`.

Classical Observational Equivalence. In a problem domain with no side-effects, observational equivalence unifies programs with the same evaluation results on the example inputs. This is expressed in the equivalence relation $\equiv_{\mathscr{E}}$ that deems two programs p_1, p_2 (observationally) equivalent if $\forall \iota \to \omega \in \mathscr{E}.\llbracket p_1 \rrbracket(\iota) = \llbracket p_2 \rrbracket(\iota)$. The OE-reduced space $(\!|\mathscr{C}|\!)^{OE}$ is then defined using $\equiv_{\mathscr{E}}$. This idea was first suggested by Udupa et al. [177], Albarghouthi et al. [6] and is in wide use today.

When performing a classical bottom-up enumeration (CBE), the vector of observed results is used as the label of the equivalence class, and each vector is only added to the bank of programs once, alongside the first program described by it, meaning any observationally equivalent programs are discarded.

A program space with side-effects. In the presence of side-effects, the space of possible programs no longer comprises only programs evaluated in the initial states $\vec{I}$. As we saw in Sec. 4.2, composition in this space requires subprograms that are evaluated in a modified state. We therefore rely on assertions to describe the states in which sub-programs are evaluated, and the states after their evaluation.

Defining the program space. The full space of programs for a set of operations $\mathscr{C}$ that includes side-effects is constructed inductively from smaller valid triples via applications of EVAL, applying the FRAME rule when necessary. We separate the terms with arity 0 into two subsets: literals *lits* and variables Vars, and for each variable $t \in \mathsf{Vars}$ denote the range of possible values for t's type as $\mathscr{K}_t$. define the full space of programs $(\!|\mathscr{C}|\!)$:

Definition 6 (Full program space)**.** We define the full program space $(\!|\mathscr{C}|\!)$ as $\bigcup_{i \geq 0} (\!|\mathscr{C}|\!)_{(i)}$ where $(\!|\mathscr{C}|\!)_{(i)}$ is defined inductively as follows:

$$
\begin{aligned}
(\!|\mathscr{C}|\!)_{(0)} =& \{\{\vec{emp}\}t\{\vec{emp};\overrightarrow{v@\bot}\} | t \in \mathit{lits}, v \in \mathscr{K} \text{ is the value of } t\} \cup \\
& \{\{\vec{\sigma}\}t\{\vec{\sigma};\langle v_i@t\rangle_i\} | t \in \mathsf{Vars}, \vec{\sigma} = \langle t \mapsto v_i\rangle_i, \vec{v} = \mathscr{K}_t \times \cdots \times \mathscr{K}_t\} \\
(\!|\mathscr{C}|\!)_{(n)} =& (\!|\mathscr{C}|\!)_{(n-1)} \cup \{\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} | \\
& \quad t \in \mathscr{C}, \mathit{arity}(t) = k > 0, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}_i';r_i\} \in (\!|\mathscr{C}|\!)_{(n-1)}, \\
& \quad \exists \vec{R}_1,\ldots,\vec{R}_k. \\
& \quad \textsc{Eval} \text{ of } t \text{ on } \{\vec{\sigma}_i * R_i\}p_i\{\vec{\sigma}_i' * R_i;r_i\} \text{ is } \{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}\ \}
\end{aligned}
$$

Finally, given $(\!|\mathscr{C}|\!)$, we can define the OE-reduced space that SOBEQ will enumerate.

4.3.4 SObEq: observational equivalence with side effects

As we showed in Sec. 4.2, classical OE's $\equiv_{\mathscr{E}}$ is no longer useful for the program space in Def. 6. We need our equivalence relation to encompass every aspect of the goal φ (Def. 5).

The formalization of OE by Peleg et al. [141] defined two properties for equivalence: *interchangeability*, i.e., two equivalent programs can be used interchangeably within a larger program, and *consistency*, i.e., two equivalent programs will either both satisfy or both not satisfy the specification.

To satisfy consistency we must include everything that participates in Def. 5. The *value* returned by the expression, and the postcondition, which should include any constrained effect. We also need the state on which the program will be evaluated to check whether it describes $\vec{I}$.

As we saw in Sec. 4.2, to satisfy interchangeability we need the *location* of the value as well: while the literal $[2,3,1]$ and a variable `x` whose valuation is $[2,3,1]$ return the same value, they are not interchangeable under a larger program like `?.sort()`, where they would lead to different postconditions. Moreover, to be able to swap a program in, its postconditions must be the same.

Combined, then, our equivalence relation simply compares the triple excluding the program:

Definition 7. (SOBEQ equivalence relation) We define the equivalence relation for $\vec{\Phi} = \langle(\iota_i \to \omega_i, q_i)\rangle_i$ to be

$$\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \equiv_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\} \iff \vec{\sigma}_1 = \vec{\sigma}_2 \wedge \vec{\sigma}'_1 = \vec{\sigma}'_2 \wedge \vec{r}_1 = \vec{r}_2$$

Or in other words, equivalence does not look at the program, only at its specifications.

Subsumption of equivalence classes. In Sec. 4.2, we considered the two triples $tr_3 = \{\texttt{arr} \mapsto [10,100,90,-1,2]\}\texttt{arr.length}\{\texttt{arr} \mapsto [10,100,90,-1,2]; 5@\bot\}$ and $tr_4 = \{emp\}\texttt{5}\{emp; 5@\bot\}$. While tr_3 and tr_4 are not observationally equivalent under Def. 7—since $emp \neq \texttt{arr} \mapsto [10,100,90,-1,2]$—$tr_4$ is *more general* than tr_3: it produces the same result ($5@\bot$) and the same effect ($\emptyset$) from more concrete states. If tr_4 can absorb tr_3, we can arrive at a more compact representation of the space.

This property of a program that differs by only a more general pre- and postcondition forms a partial order.

Definition 8 (Generality ordering of triples)**.** Given a specification $\vec{\Phi}$, we define

$$\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\} \iff \exists\vec{R}.\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \equiv_{\vec{\Phi}} \{\vec{P}_2 * \vec{R}\}p_2\{\vec{Q}_2 * \vec{R};\vec{r}_2\}$$

In other words, the larger program is the more general way to achieve the same result and the same effect, as in the order of *predicate abstraction* [62].

While $\sqsubseteq_{\vec{\Phi}}$ is clearly not an equivalence relation (the existence of a frame axiom is not symmetrical) so it cannot be used to partition the space into classes, we notice that replacing a triple with one that is larger according to $\sqsubseteq_{\vec{\Phi}}$ does preserve the two properties of an OE-relation, *interchangeability* and *consistency*, in one direction. Originally [141], *interchangeability* states that if two programs are equivalent, replacing them within a larger program will result in an equivalent larger program, and *consistency* states that given a specification φ, if two programs are equivalent, either both satisfy φ or neither one does. Under $\sqsubseteq$, however, a slightly weaker—but still useful—version holds.

Interchangeability under $\sqsubseteq_{\vec{\Phi}}$ means that given $t \in \mathscr{C}$ with $arity(t) = k$, and k triples $tr_j = \{\vec{P_j}\}p_j\{\vec{Q}_j;\vec{r}_i\}, 1 \leq j \leq k$, and $\vec{R}_i$ s.t. EVAL can be applied on t and $\{\vec{P}_j * \vec{R}_j\}p_j\{\vec{Q}_j * \vec{R}_j;\vec{r}_j\}$ to infer $\{\vec{P}\}t(p_1,\dots,p_k)\{\vec{Q};\vec{r}\}$, and a triple $tr'_i = \{\vec{P}'_i\}p'_i\{\vec{Q}'_i;\vec{r'}_i\}$ s.t. $tr_i \sqsubseteq_{\vec{\Phi}} tr'_i$ for some $1 \leq i \leq k$, then

1. There exists $\vec{R'}_i$ s.t. EVAL can still be applied on t when replacing $\{\vec{P}_i * \vec{R}_i\}p_i\{\vec{Q}_i * \vec{R}_i;\vec{r}_i\}$ with $\{\vec{P'}_i * \vec{R}'_i\}p'_i\{\vec{Q'}_i * \vec{R}'_i;\vec{r'}_i\}$ to infer $\{\vec{P'}\}t(p_1,\dots,p'_i,\dots,p_k)\{\vec{Q}';\vec{r}'\}$, and
2. $\{\vec{P}\}t(p_1,\dots,p_k)\{\vec{Q};\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}t(p_1,\dots,p'_i,\dots,p_k)\{\vec{Q}';\vec{r}'\}$

Consistency under $\sqsubseteq_{\vec{\Phi}}$ means that for goal $\mathscr{G}$, if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, then if there exists $\vec{R}_1$ s.t. $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathscr{G}$, then there exists $\vec{R}_2$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2;\vec{r}_2\}$ matches $\mathscr{G}$. It's important to notice that the $\Leftrightarrow$ of the equivalence version of *consistency* is now $\Rightarrow$: if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\}$ is a result, so is $\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, but since $\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$ is more general $\vec{P}_2$ might cover ι even if $\vec{P}_1$ did not.

Lemma 4.3.1. *Interchangeability* and *consistency* are preserved under $\sqsubseteq$.

The proof of Theorem 4.3.1 is found in Sec. 4.A.1.

With the program space $(\!|\mathscr{C}|\!)$ and the SOBEQ equivalence relation and generality ordering, we are ready to search for a program. The next section will construct the OE-reduced space via enumeration.

4.4 The SObEq Enumeration

In this section, we describe the construction of the OE-reduced space. We first describe the structure of the space, then the enumeration algorithm and specifically selecting children for applied terms, and finally, we describe how the enumeration can find equivalence classes that, by virtue of their generality, subsume other equivalence classes.

Classical Bottom-Up Enumeration Algorithm. A bottom-up enumeration consists of applying each $t \in \mathscr{C}$ to k-tuples of child programs from the bank of previously seen programs, where $arity(t) = k$. Each newly enumerated $t(p_1, \ldots, p_k)$ is then evaluated, and if an equivalence class labeled by the vector of its observed values is not yet represented in the bank, it is added to the bank. The enumeration of programs is done according to some ordering, e.g., height, number of AST nodes, or probability score [14]. This would usually be facilitated by the structure of the bank to allow for easier retrieval of relevant child-programs.

The precondition bank. In our setting, we want to bank our discovered triples according to their precondition: for each vector of assertions $\vec{a}$ we keep a separate bank for all the discovered programs whose precondition vector is $\vec{a}$.

There are two reasons for this: first, when composing a k-tuple of arguments to t, finding a sequence that EVAL is enabled for can be performed constructively rather than by filtering all possible k-tuples. For each sequence of length $k-1$ with postconditions $\vec{Q}_{k-1}$, the enumeration can preemptively rule out any next child options from assertion vectors $\vec{a}$ that will never form a valid sequence, i.e., have different values assigned to the variables appearing in both $\vec{Q}_{k-1}$ and $\vec{a}$. In algorithm 1 we call this construction COLLECTCHILDREN.

Second, and even more importantly, this allows us to track what assertions have been discovered by the enumeration. If the enumeration produces a triple $\{\vec{a}\}p\{\vec{a}';\vec{r}\}$ where $\vec{a}'$ has not yet been seen, then a new bank will be initialized for $\vec{a}'$.

Initializing a new bank. When a new assertion $\vec{a}$ is discovered by the enumeration, two things happen: first, if $\vec{a}$ was discovered as a precondition, a new section of the bank is constructed for $\vec{a}$. Second, even if $\vec{a}$ does not have a triple ready to add to it (i.e., it was discovered as a postcondition), we add all variables that can be used in $\vec{a}$ to the bank. To preserve the invariant that all programs are banked with their footprint, we construct and add $\{a_x\}x\{a_x;r_x\}$ where

$a_x = \langle x \mapsto a_i(x) \rangle_i$ and $r_x = \langle a_i(x) \rangle_i$ for each x in $\vec{a}$, also initializing a bank for a_x if needed.

In algorithm 1, both steps of the initialization together are denoted $\textsc{InitBank}(\vec{a})$.

At the beginning of the enumeration, the bank is trivially initialized with the empty assertion $\vec{emp}$ for literals and with $\langle x \mapsto \iota_i(x) \rangle_i$ for each variable x in the input.

Enumerating terms. The main enumeration is not very different from the CBE enumeration: the enumerator selects $t \in \mathscr{C}$ with $arity(t) = k > 0$, and composes as arguments for it all k-tuples of triples $\{\vec{P_i}\}p_i\{\vec{Q}_i;\vec{r}_i\}$ that $\textsc{Eval}$ can be applied to with t from triples already in the bank.

The application of $\textsc{Eval}$ then constructs a new triple $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ that can then be *observed*: we use its pre- and postcondition assertions and its results as the label for its equivalence class. This label is searched for in the bank for precondition $\vec{P}$. If the bank for $\vec{P}$ has an equivalent program, $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ is discarded, and if no equivalent program was previously seen, $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ is added to the bank for $\vec{P}$.

However, because we also have at our disposal $\sqsubseteq_{\vec{\Phi}}$ (Def. 8), we can do even better.

Using more general triples. Under the order relation $\sqsubseteq_{\vec{\Phi}}$, more general triples can subsume less general ones in the course of enumeration, even though they are not strictly equivalent. However, we must consider two cases where this can happen: (i) the less general triple being enumerated after the more general triple is in the bank, and (ii) a more general triple being enumerated later.

Handling (i) is simple enough: when the enumerator enumerates $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, even if there is no representative for the equivalence class of $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, the enumerator checks whether some $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$ s.t. $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$ already exists in the bank, and if it does, discards $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$. This is efficiently done by searching the bank with the label a more general program would have, each time replacing the assertion components in the label for all $\vec{a}$ in the bank that are more general than $\vec{\sigma}$.

Handling (ii), however, is a larger feat. *Theoretically*, once we enumerate the more generic triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, we can find all less-general triples $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$ and replace them

```
Algorithm 1: The SOBEQ enumeration
    Input  : Set of specifications $\vec{\Phi} = \langle(\iota_i \to \omega_i, q_i)\rangle_i$, variables Vars specified in
             example inputs
    Output: Program $p$ composed of components $\mathscr{C}$ s.t. that satisfies $\vec{\Phi}$
    Globals: Precondition bank $bank$, mapping from assertion to an OE bank
 1  initialAsserts := $\{e\vec{m}p\} \cup \{\langle x \mapsto \iota_i(x)\rangle_i \mid x \in \mathsf{Vars}\}$
 2  foreach $\vec{a} \in initialAsserts$ do INITBANK($\vec{a}$);
 3  repeat // until timeout
 4      foreach $t \in \mathscr{C}, k = arity(t) > 0$ do
 5          foreach $\{\vec{\sigma}_1 * R_1\}p_1\{\vec{\sigma}'_1 * R_1;\vec{r}_1\} \cdots \{\vec{\sigma}_k * R_k\}p_k\{\vec{\sigma}'_k * R_k;\vec{r}_k\} \in$
             COLLECTCHILDREN($bank, k$) do
 6              $\{\vec{\sigma}\}p\{\vec{Q};\vec{r}\}$ := EVAL
                 $(t, \{\vec{\sigma}_1 * R_1\}p_1\{\vec{\sigma}'_1 * R_1;\vec{r}_1\} \cdots \{\vec{\sigma}_k * R_k\}p_k\{\vec{\sigma}'_k * R_k;\vec{r}_k\})$
 7              if $\exists \vec{R}.\{\vec{\sigma} * \vec{R}\}p\{\vec{Q} * \vec{R};\vec{r}\}$ matches $\mathscr{G}_{\vec{\Phi}}$ then return $p$;
 8              if $\forall \vec{a} \in keys(bank).\exists \vec{R}.(\vec{a} * \vec{R} = \vec{\sigma} \wedge \vec{R} \subseteq \vec{Q}) \Rightarrow (\vec{a}, \vec{Q} \setminus \vec{R}, \vec{r}) \notin bank(\vec{a})$
                 then
 9                  if $\vec{\sigma} \notin keys(bank)$ then INITBANK($\vec{\sigma}$);
10                  $bank(\vec{\sigma})$ += $\{\vec{\sigma}\}p\{\vec{Q};\vec{r}\}$
11              end
12              if $\vec{Q} \notin keys(bank)$ then INITBANK($\vec{Q}$);
13          end
14      end
15  until timeout;
```

with $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$: removing them from the bank, and replacing p_2 with p in any compositions. But if this was not expensive enough on its own, this is where Theorem 4.3.1 comes into play: this replacement can weaken the pre- and postconditions of the compositions, causing a cascade of weakening more and more triples in the bank. Somewhere during this cascade, a triple that did not satisfy $\vec{\Phi}$ because $\vec{P}$ did not describe $\vec{I}$ might have its precondition weakened such that the new precondition does describe $\vec{I}$, turning an already-enumerated program into a solution. When even just the replacement of a single triple is expensive (see Sec. 4.7.5), such a cascade operation is downright prohibitive. As such, we elect to only incorporate (i) into SOBEQ.

4.4.1 The enumeration algorithm

We can now compose the complete SOBEQ enumeration, seen in algorithm 1.

Lines 1–2 of the algorithm initialize the enumeration by calling InitBank, which operates as described above, to create the basic programs for $\vec{emp}$ and for each of the variables' initial values.

Line 4 selects the next component for application. Components with an arity of 0 do not need to be considered, as they are handled exclusively by InitBank. Next, on line 5 CollectChildren iteratively constructs all k-tuples of children Eval is enabled for by limiting the preconditions from which the next child can be selected. CollectChildren returns each k triples already updated with the frame axioms $R_1, \ldots, R_k$ to a unified composed heap.

Line 6 then applies Eval to construct the new triple, and if it is a solution to the synthesis task it will be returned by line 7. If not, line 8 checks whether an equivalent or subsuming program already exists in the bank via its equivalence class label: notice that precondition $\vec{a}$ in the bank is deemed relevant if there exists a frame axiom $\vec{R}$ s.t. $\vec{R}$ both matches $\vec{a}$ to $\vec{\sigma}$ and all values in $\vec{R}$ are unchanged in $\vec{Q}$ (i.e., $\vec{R}$ truly is a frame). Every $\vec{a}$ in the bank lookup uses the relevant equivalence class label, with $\vec{a}$ as its precondition and subtracting the frame axiom from its postcondition assertion. If no triple is found, the new triple is added to the bank. If either the pre- or postcondition assertion do not exist in the bank, InitBank is called for them to generate triples for their variables.

4.5 Correctness of SObEq

In this section, we present the correctness theorems of both the SObEq enumeration and SObEq's reduction of the space. We begin with the enumeration of the space.

4.5.1 Correctness of the SObEq enumeration

Def. 6 defines a space $(\!|\mathscr{C}|\!)$ where $(\!|\mathscr{C}|\!)_{(0)}$ starts with all possible values of each variable v, and, inductively, will cover all possible concrete states. Our enumeration, however, starts from $\langle x \mapsto \iota_i(x)\rangle_i$, i.e., only from the variable values in the inputs, and only enumerates programs using other preconditions as it discovers them. Many $\vec{\sigma} \in \mathscr{A} \times \cdots \times \mathscr{A}$ will simply not be reached by the enumeration.

We therefore define the *reachable* program space (first without, and later with the observational equivalence reduction), and introduce a correctness theorem for its enumeration, i.e., that it will not lose a solution that exists in the full program space.

Definition 9 (Reachable Program Space)**.** We define a reachable program space from some initial states $\vec{I}$ to be $(\!|\mathscr{C}|\!)^{reach} = \bigcup_{i\geq 0} (\!|\mathscr{C}|\!)^{reach}_{(i)}$, where

$$\begin{aligned}
(\!|\mathscr{C}|\!)^{reach}_{(0)} &= \{\{\vec{emp}\}t\{\vec{emp};\overrightarrow{v@\bot}\} | t \in lits, v \in \mathscr{K} \text{ is the value of } t\} \\
&\quad \cup \big\{\{\vec{\sigma}\}t\{\vec{\sigma};\vec{r}\} | t \in \mathsf{Vars}, \vec{\sigma} = \langle t \mapsto \iota_i(t)\rangle_i, \vec{r} = \langle \iota_i(t)@t\rangle_i\big\} \\
(\!|\mathscr{C}|\!)^{reach}_{(n)} &= (\!|\mathscr{C}|\!)^{reach}_{(n-1)} \\
&\quad \cup \big\{\{\vec{P}\}t(p_1,\dots,p_k)\{\vec{Q};\vec{r}\} | t \in \mathscr{C}, arity(t) = k > 0, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\in\}(\!|\mathscr{C}|\!)_{(n-1)}, \\
&\qquad \exists \vec{R}_1,\dots,\vec{R}_k.\textsc{Eval} \text{ of } t \text{ on } \{\vec{\sigma}_i * R_i\}p_i\{\vec{\sigma}'_i * R_i; r_i\} \text{ is } \{\vec{P}\}t(p_1,\dots,p_k)\{\vec{Q};\vec{r}\}\big\} \\
&\quad \cup \big\{\{\vec{a}_t\}t\{\vec{a}_t;\vec{r}\} | t \in \mathsf{Vars}, \vec{a}_t = \langle t \mapsto \sigma_i(t)\rangle_i, \vec{r} = \langle \sigma_i(t)@t\rangle_i, \\
&\qquad \{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r'}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n-1)} \vee \{\vec{\sigma}'\}p\{\vec{\sigma};\vec{r'}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n-1)}\big\}
\end{aligned}$$

Theorem 4.5.1. If a valid solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ exists in $(\!|\mathscr{C}|\!)$, then $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}$.

In Sec. 4.A.2, we prove Theorem 4.5.1 by showing that *all* programs that can be reached in a sequence from the initial state in$(\!|\mathscr{C}|\!)$ also exist in $(\!|\mathscr{C}|\!)^{reach}$.

4.5.2 Correctness of the SObEq reduction

Next, we show that the SOBEQ enumeration (algorithm 1) is correct: its reduction will not lose any solutions.

Definition 10 (SObEq-Reduced Program Space)**.** Given the specifications $\vec{\Phi}$ and the equivalence relation $\equiv_{\vec{\Phi}}$ (Def. 7) and order relation $\sqsubseteq_{\vec{\Phi}}$ (Def. 8) they induce, we define the reduction

$$\begin{aligned}
reduce(\vec{\Phi}, \mathscr{S}_{-1}, \mathscr{S}) = \big\{s(c) = \{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \mid c \in \mathscr{S}/\equiv_{\vec{\Phi}}, \\
\forall \{\vec{P}\}p'\{\vec{Q};\vec{r'}\} \in \mathscr{S} \cup \mathscr{S}_{-1}.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \not\sqsubseteq_{\vec{\Phi}} \{\vec{P}\}p'\{\vec{Q};\vec{r'}\}\big\}
\end{aligned}$$

where $s(c)$ selects a representative of equivalence class c, and $/ \equiv_{\vec{\Phi}}$ partitions the space into equivalence classes based on $\equiv_{\vec{\Phi}}$. We then define the space after the SOBEQ reduction to be $\bigcup_{i \geq 0} (\!|\mathscr{C}|\!)^{S}_{(i)}$ where $(\!|\mathscr{C}|\!)^{S}_{(i)}$ is also defined inductively:

$$
\begin{aligned}
(\!|\mathscr{C}|\!)^{S}_{(0)} &= \mathit{reduce}(\vec{\Phi}, \emptyset, (\!|\mathscr{C}|\!)^{reach}_{(0)}) \\
(\!|\mathscr{C}|\!)^{S}_{(n)} &= (\!|\mathscr{C}|\!)^{S}_{(n-1)} \cup \mathit{reduce}\Big(\vec{\Phi}, (\!|\mathscr{C}|\!)^{S}_{(n-1)}, \{ \\
&\qquad \{\vec{P}\}t(p_1, \ldots, p_k)\{\vec{Q}; \vec{r}\} | t \in \mathscr{C}, \mathit{arity}(t) = k > 0, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i; \vec{r}_i\} \in (\!|\mathscr{C}|\!)^{S}_{(n-1)}, \\
&\qquad \exists \vec{R}_1, \ldots, \vec{R}_k . \textsc{Eval} \text{ of } t \text{ on } \{\vec{\sigma}_i * R_i\}p_i\{\vec{\sigma}'_i * R_i; r_i\} \text{ is } \{\vec{P}\}t(p_1, \ldots, p_k)\{\vec{Q}; \vec{r}\}\} \\
&\qquad \cup \big\{ \{\vec{a}_t\}t\{\vec{a}_t; \vec{r}\} | t \in \mathsf{Vars}, \vec{a}_t = \langle t \mapsto \sigma_i(t) \rangle_i, \vec{r} = \langle \sigma_i(t)@t \rangle_i, \\
&\qquad \{\vec{\sigma}\}p\{\vec{\sigma}'; \vec{r}'\} \in (\!|\mathscr{C}|\!)^{S}_{(n-1)} \vee \{\vec{\sigma}'\}p\{\vec{\sigma}; \vec{r}'\} \in (\!|\mathscr{C}|\!)^{S}_{(n-1)} \big\} \Big)
\end{aligned}
$$

Then to show that the SOBEQ enumeration using $\equiv_{\vec{\Phi}}$ and $\sqsubseteq_{\vec{\Phi}}$ is correct, we re-introduce the correctness theorem [141, Theorem 7.2] for an OE-reduced enumeration in our terms:

Theorem 4.5.2 (Correctness of the reduced enumeration). Given $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1; \vec{r}_1\} \in (\!|\mathscr{C}|\!)^{reach}$ and a goal $\mathscr{G}_{\vec{\Phi}}$, if there exists $\vec{R}_1$ s.t. $\{\vec{\sigma}_1 * \vec{R}_1\}p_1\{\vec{\sigma}'_1 * \vec{R}_1; \vec{r}_1\}$ matches $\mathscr{G}_{\vec{\Phi}}$, then there exists $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2; \vec{r}_2\} \in (\!|\mathscr{C}|\!)^{S}$ and a $\vec{R}_2$ s.t. $\{\vec{\sigma}_2 * \vec{R}_2\}p_2\{\vec{\sigma}'_2 * \vec{R}_2; \vec{r}_2\}$ also matches $\mathscr{G}_{\vec{\Phi}}$.

The detailed proof is in Sec. 4.A.3.

4.6 Implementation

Our implementation of the SOBEQ algorithm (algorithm 1) is a synthesizer for JavaScript programs implemented in Scala. The component set $\mathscr{C}$ contains 54 components for integers, strings, arrays, and sets, including nine mutating operations: postfix increment for integers, the array functions `sort`, `reverse`, `splice`, `push`, `pop`, `shift`, and the set functions `add` and `delete`.[2] $\mathscr{C}$ also includes the variables from $\mathscr{E}$ and a sequence operator for any two expressions. We allow a task to include additional string constants provided by benchmarks from the literature [9, 14] or interaction models [52, 51].

[2]JavaScript strings are immutable, so $\mathscr{C}$ includes no mutating components for strings.

Preserving immutability. The effect constraints in $\vec{\Phi}$ can specify that a variable has not changed in the postcondition of the solution $\vec{\sigma}'$. But because we only reason about the pre- and postcondition of a triple, not the path between them, effect constraints cannot distinguish between, e.g., `x` and `--(++x)`. This is generally desirable as it marks `--(++x)` as redundant, but this will only be discovered when applying `--`. If the user wants `x` to *always* be immutable, we can do better.

We add to our task specifications the ability to mark a variable as `immut`. Once the user has asked that `x` be immutable, there is no reason to construct `++x` at all: a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ where $\langle\sigma_i(x)\rangle_i \neq \langle\sigma_i'(x)\rangle_i$ is discarded immediately, and $\vec{\sigma}'$ is not initialized in the bank. Any programs with the precondition $\vec{\sigma}'$ assume a mutation of `x`, so if `x` is marked as `immut`, eliminating the entire precondition entirely is correct.

4.7 Experimental Evaluation

We empirically evaluate SOBEQ on our implementation. All our experiments used a server with Intel Xeon Gold 6338 2 GHz[3] and 128GB of RAM, with the JVM maximum heap size set to 110 GB, and a timeout of one hour.

Benchmarks. We evaluated SOBEQ on a set of 46 benchmarks, curated from previous synthesizers [161, 14, 51], programming exercises on LEETCODE.COM, and questions on STACKOVERFLOW.COM. We split the benchmarks into two sets: ▷ ***Mutating***: 10 benchmarks *requiring* mutation to be solved in $\mathscr{C}$. ▷ ***Pure***: 36 benchmarks that can be solved by pure expressions.

Research Questions. We aim to answer the following research questions:

RQ1) Can SOBEQ solve a variety of synthesis tasks correctly?

RQ2) How does SOBEQ compare to the state-of-the-art in enumerative synthesis with side effects?

RQ3) What impacts the size of the space discovered by SOBEQ?

RQ4) What is the overhead of SOBEQ when mutation is prohibited?

[3]This processor has 32 cores/64 threads, but our implementation is single-threaded.

RQ5) How does subsumption of equivalence classes affect enumeration?

4.7.1 RQ1: Benchmark results

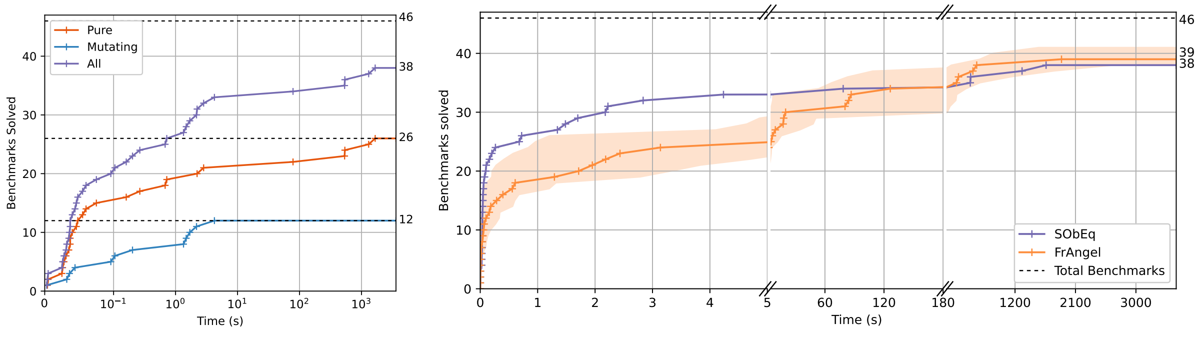
Our implementation successfully solved 38 (83%) of the benchmarks, 8 (80%) from the *Mutating* set, and 30 (83%) from the *Pure* set. 42% of the benchmarks were solved in less than 5 seconds. Excluding timeouts, the benchmarks were solved in an average of 1m 49s (SD=352s), and median time of 0.09s. Fig. 4.5a plots benchmarks solved over time according to the components used in the solution, i.e., *Pure* benchmarks with mutating solutions are counted as mutating. We see that mutating solutions are found as quickly as pure ones, and in fact all mutating solutions are found under 5 seconds.

We manually inspected the solution to each benchmark to check if the solution is overfitted to the examples in $\vec{\Phi}$, and if it contains spurious mutations which are not required to satisfy $\vec{\Phi}$. One benchmark had an overfitted solution, and four of the *Pure* benchmarks were solved with a non-*Pure* solution. Three of these used a `splice` function to find a shorter solution. The other used `push`, which returns the new length of the array, to avoid using the larger `array.length + 1`.

Overall, **we answer RQ1 in the *affirmative*.** SOBEQ successfully solved the vast majority of the benchmarks with generalizable solutions, and without spurious mutations.

4.7.2 RQ2: Comparison to FRANGEL

We compare SOBEQ to FRANGEL [161], a stochastic synthesizer considered the state of the art in synthesizing general programs with mutations. We defined a custom grammar for FRANGEL that contains the same set of components as our implementation, and disabled the enumeration of loops, conditionals, and assignments. FRANGEL's enumeration is not deterministic, so we ran FRANGEL five times on each of the 46 benchmarks, and compared the best, worst and median performance to SOBEQ in Fig. 4.5b. We use FRANGEL's median times for the rest of this analysis.



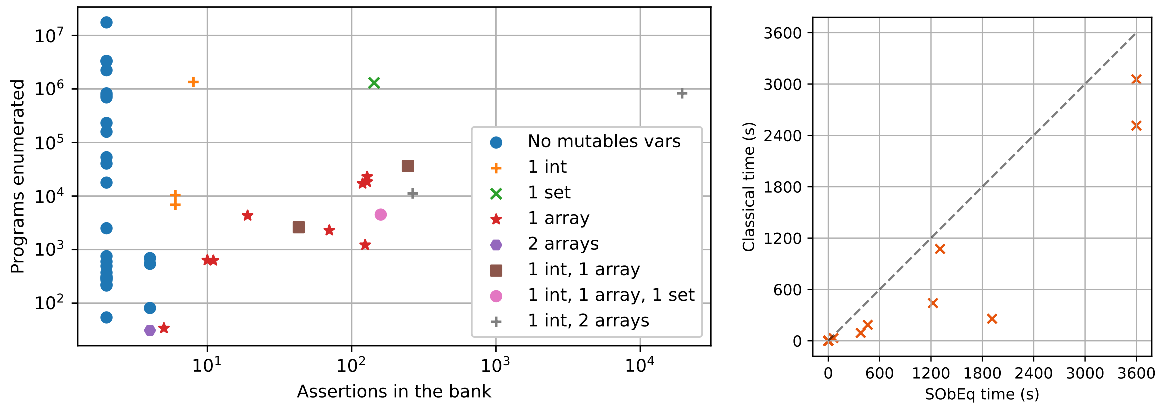
(a) RQ1: Performance of SOBEQ **(b)** RQ2: Comparison to FRANGEL

Figure 4.5. (a) The number of benchmarks SOBEQ solves over time, split into benchmarks solved with mutating components and benchmarks solved only with pure components. (b) No. of benchmarks solved over time. FRANGEL is stochastic, so its line shows median values with the range showing the maximum and minimum.

SOBEQ is able to solve more benchmarks (32 vs. 24) in the first five seconds, and shows comparable performance overall, with FRANGEL having the same mean time of 1m 49s (SD=329s). Performance on individual benchmarks varies, with SOBEQ faster on 24, and FRANGEL on 18 benchmarks, including four benchmarks which SOBEQ solves in under two seconds but FRANGEL times out on, and three benchmarks which FRANGEL solves in under two minute but SOBEQ times out on. This is not surprising given the difference in enumeration techniques. We also inspected the solutions manually, and found that FRANGEL had more overfitted solutions, with four benchmarks with overfitted solutions in all five runs, and another four overfitting in at least one run. FRANGEL only had one spurious mutation in all its runs. **To answer RQ2, SOBEQ shows comparable performance to FRANGEL, while being deterministic, and finding fewer overfitted solutions.**

4.7.3 RQ3: The size of the search space

We inspected the precondition bank at the end of each run from Sec. 4.7.1 to see how many assertions were discovered. We found that the number of precondition in the bank varied widely across different benchmarks. Aside from the total number of programs enumerated, the main determinant of the number of assertions was the type of variables defined in each



(a) RQ3: Size of the space **(b)** RQ4: Comparison with CBE

Figure 4.6. (a) No. of preconditions in the bank at termination/timeout, compared with total no. of programs enumerated for that benchmark. The shape and color of each point indicates the number of mutable variables available in that benchmark's $\vec{\Phi}$. (b) Time to termination/timeout for each *Pure* benchmark with SOBEQ, compared to a classical bottom-up enumeration with OE. Points at (timeout,timeout) not shown.

benchmark's example inputs. $\mathscr{C}$ has a varying number of mutating components for each type (one for integers, two for sets, and six for arrays) and these had a significant effect on the enumerated space. More specifically, benchmarks with only string and boolean variables did not discover any new assertions outside the initial preconditions and their compositions, while the rest discovered assertions depending on the number of mutating components for each variable type. As Fig. 4.6a shows, benchmarks with one integer discover modestly more assertions than those with none, those with an array form a far steeper slope, and the benchmark with one set falls in between. **To answer RQ3, the size of the search space is largely determined by the number of mutating components in $\mathscr{C}$.**

In addition, Fig. 4.7 shows the distribution of precondition sizes for each of our benchmarks.

4.7.4 RQ4: Overhead of SOBEQ

If $\mathscr{C}$ does not have any mutating components, SOBEQ would not discover any assertions outside the initial preconditions and their compositions. This reduces it to a classical bottom-up enumeration in terms of the space of programs it describes. In this experiment we measured what happens *in practice*: the more fine-grained equivalence classes resulting from using the result rather than just the value in the equivalence class labels, and the additional overhead from creating a compound heap for sub-programs with different variables, means that our implementation will have an overhead compared to a traditional bottom-up synthesizer.

To measure the overhead, we restricted our $\mathscr{C}$ to just the non-mutating components, and compared its performance on *Pure* benchmarks against a version that enumerates all programs in the initial concrete state, and uses the classical value-based equivalence relation for an OE-reduction. This version uses the same enumeration algorithm as SOBEQ, but with all programs enumerated and evaluated on the initial concrete state. Thus it is equivalent to classical OE with height-based enumeration. By removing the mutating components for this evaluation, we avoid the correctness issues in classical OE, while controlling for performance differences caused by having different grammars.

As expected, in this restricted context CBE significantly outperforms SOBEQ (time comparison shown in Fig. 4.6b). Classical enumeration solved the benchmarks an average of 2.59 times faster (Median=2.52), and it was able to solve two additional benchmarks in the one hour timeout.

Overhead aside, we found that another reason for this difference in performance is the enumeration order. Unlike classical OE, which prioritizes children by the order that they were enumerated, SOBEQ adds an external order by assertions. While this does not have as large an impact in the early iterations, as the bank grows this can lead to large differences in time, and to a different solution being found first. For instance, for the outlier benchmark in Fig. 4.6b with SOBEQ time of 1900s, SOBEQ was not only 7 times slower, but it found a solution with a different subprogram. For this, and the two benchmarks which only classical solved within the timeout, the solution is of height 4, by which point the exponential nature of the search space means that different child enumeration orders can have a large effect on performance. **For RQ4, we found that SOBEQ has a significant overhead compared to classical enumeration,**

caused by the more fine-grained equivalence classes, the overhead of creating compound heaps, and the different enumeration order.

4.7.5 RQ5: Ablation of subsumption of equivalence classes

We performed an ablation of using $\sqsubseteq_{\vec{\Phi}}$ instead of $\equiv_{\vec{\Phi}}$ in the enumeration. As noted in Sec. 4.4, a more general triple can be enumerated before or after its less general counterpart. To investigate the impact of possible actions, we tested SOBEQ in three configurations: (1) *Equiv*: only use $\equiv_{\vec{\Phi}}$, i.e., add less general triples to the bank, (2) *Discard*: use $\sqsubseteq_{\vec{\Phi}}$ to discard less general triples when more general ones exist in the bank, i.e., SOBEQ's behavior, and (3) *Replace*: use $\sqsubseteq_{\vec{\Phi}}$ to discard less general triples whether they are newly enumerated or already in the bank. The *Replace* configuration requires replacing the program discarded from the bank, but does not perform the cascading operation described in Sec. 4.4.

We found that subsumption did not have a large effect on performance, but that overall the *Discard* configuration was the fastest. Compared to it, *Equiv* solved the benchmarks an average of 1.09 times slower (SD=0.36), and *Replace* solved them an average of 1.02 times slower (SD=0.14). The main reason for the small effect size is that only a small number of enumerated programs are subsumed, with a mean of 3.60% of programs discarded (SD=4.32%, Median=1.86%), and only 1.93% of programs replaced (SD=3.04%, Median=0.31%). Because of this, the overhead of subsumption was competing against the small benefits, with the cheaper discard operation being effective, while the more expensive replace operation was not. **To answer RQ5, subsumption of equivalence classes has a small effect on performance, with the cheaper discard operation being the most effective.**

4.8 Related work

Syntax- and Component-Guided Synthesis. Syntax-Guided Synthesis (SyGuS) [7] is a form of the synthesis problem providing a specification along with a grammar defining its program space. It can also describe a form of synthesis where programs are constructed by syntax rules.

The latter is essentially identical to Component-Based Synthesis, where the synthesizer is provided with a list of functions and operations spanning the program space. Both of these formulations are tackled by many synthesis strategies: enumerative [103, 161, 14, 48, 70, 57, 82, 164, 25, 198, 102], SMT [88, 12, 150, 3] and SAT [167, 172] based, by traversing constructed representations of the space such as VSAs [89, 99, 199], version spaces [163] and Petri-nets [49, 69].

Proof-directed synthesizers. A specific form of enumerative synthesis is one that attempts to create a proof for a proof goal, where the resulting program follows the proof. This idea is not new [170], and some deductive type-driven synthesizers describe their program search as a proof search [69, 135]. However, other works' goals are more expressive than types: refinement types [146], relational specifications [93], select/update operators [128], Separation Logic, and even natural language [34]. Importantly, proof searches are overwhelmingly *deductive*, or perform a top-down search of the proof space. To our knowledge, SOBEQ is the first synthesis proof search that enumerates the proof space bottom-up.

Synthesis with Separation Logic. Separation Logic proofs are a popular vehicles for proof-directed synthesis. They have been successfully used for parallelizing non-parallel code [21], but are most often used to synthesize heap-manipulating programs. SUSLIK [147] performs a deductive search with inferences about the symbolic heap to generate programs performing pointer operations. CYPRESS [84] extends this with the ability to synthesize recursive auxiliaries, ROBOSUSLIK [35] extends it with read-only borrows, and RUSSOL [55] applies it to Rust programs, leveraging Rust's type system to simplify the required specification from formal assertions to functional annotations of Rust functions. All these tools accept assertions as their specification, hinge on manually-crafted inference rules, and employ solvers. SOBEQ, in contrast, requires only specifications of variable and result values, and does not require a solver, only an interpreter.

Observational equivalence. Observational Equivalence [177, 6] is a popular way to reduce the size of the space when enumerating bottom-up. The original formulation of OE is for a specification by examples, but RESL [141] generalized OE to encompass other specifications on programs as well; this generalization also introduced the notions of *interchangeability* and *consistency* that SOBEQ's correctness relies on. SYMETRIC [53] relaxes OE to observational similarity by using clustering rather than equivalence to choose a representative, then performing repair on the resulting program. While this unifies equivalence classes like SOBEQ's subsumption, correctness of SYMETRIC's solutions hinges on the success of repairing its intermediate result.

Many of the synthesizers that use OE only synthesize pure programs [142, 14, 141, 6, 185, 127] (we include [141] in this list, despite its misuse of `Array.sort` being part of the motivation for this work), others employ synthesis with OE to produce pure expressions that are then used in assignments or other larger mutating expressions [113, 114, 177, 52, 51], creating a top-level program with side effects. Two interesting approaches to deal with effects under OE are LENS [145] and ARBORIST [108]. ARBORIST synthesizes calls to `fold` by iteratively refining the space of reachable intermediate states, terminating only if this refinement terminates. LENS synthesizes assembly code by discovering all intermediate states in a reduced state-space, then continuously refining that search. Both hinge on the ability to unify equivalent programs *without* discarding any programs using an OE implementation that does not use a bank like SOBEQ does, and which is only tractable under very strict assumptions.

Synthesis with side effects. The problem of synthesizing mutating snippets has been worked around in many ways. SL proof-directed approaches assume an extremely limited target language of mutations that can be described by the proof. SKETCH[167], TRANSIT [177], SNIPPY [52], SIMPL [165] and COZY [113, 114] assume expressions are pure, then use them in specific mutating contexts validated by other means (e.g., SMT). SYPET [49] and FRANGEL [161] rely on other means for getting programs—Java APIs' rich type systems and drawing them at random, resp.—and only then evaluate them. LOOPY [51] works in a domain where intermediate states caused by mutations are bounded and enumerates paths between these states; similarly

LENS [145] reduces all registers to 8-bit in order to construct the full space of intermediate states. RBSYN [70] performs a type-guided top-down enumeration from a syntax including *effect holes* that can be filled with annotated mutating methods; expressions not within effect holes have a purity assumption, limiting how effectful components are composed, unlike SOBEQ that can use mutating operations anywhere. COBALT [128] also separates pure and impure expressions, using a conflict-driven enumeration to search for a proof matching its goal using provided API annotations. CODEHINT [57] has the most elaborate solution, evaluating and then undoing in-memory side effects of candidate expressions. SOBEQ sheds the assumptions that expressions are pure, that intermediate states are known in advance, and that annotations guide the search. Moreover, SOBEQ comes with a formal guarantee unlike the two most general previous solutions: FRANGEL, which randomly draws programs, and CODEHINT, which considers only the first provided example when searching for candidate programs.

4.9 Acknowledgements

Chapter 4, in part, is currently being prepared for submission for publication of the material. It was done in collaboration with Hila Peleg. The dissertation author was a primary investigator and author of this material.

4.A Proofs of Theorems and Lemmas

4.A.1 Proof of Theorem 4.3.1

Interchangeability under $\sqsubseteq_{\vec{\Phi}}$ means that given $t \in \mathscr{C}$ with $arity(t) = k$, and k triples $tr_j = \{\vec{P}_j\}p_j\{\vec{Q}_j;\vec{r}_i\}, 1 \leq j \leq k$, and $\vec{R}_i$ s.t. EVAL can be applied on t and $\{\vec{P}_j * \vec{R}_j\}p_j\{\vec{Q}_j * \vec{R}_j;\vec{r}_j\}$ to infer $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$, and a triple $tr'_i = \{\vec{P}'_i\}p'_i\{\vec{Q}'_i;\vec{r'}_i\}$ s.t. $tr_i \sqsubseteq_{\vec{\Phi}} tr'_i$ for some $1 \leq i \leq k$, then

1. There exists $\vec{R'}_i$ s.t. EVAL can still be applied on t when replacing $\{\vec{P}_i * \vec{R}_i\}p_i\{\vec{Q}_i * \vec{R}_i;\vec{r}_i\}$ with $\{\vec{P'}_i * \vec{R}'_i\}p'_i\{\vec{Q'}_i * \vec{R}'_i;\vec{r'}_i\}$ to infer $\{\vec{P'}\}t(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$, and
2. $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}t(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$

Proof: interchangeability. Because $tr_1 \sqsubseteq_{\vec{\Phi}} tr'_1$, there exists $\vec{R}'$ s.t. $\{\vec{P}'_i * \vec{R}'\}p'_i\{\vec{Q}'_i * \vec{R}';\vec{r'}_i\} \equiv_{\vec{\Phi}} \{\vec{P}_i\}p_i\{\vec{Q}_i;\vec{r}_i\}$. Therefore, let us select $\vec{R}'_i = \vec{R}_i * \vec{R}'$, which will keep the composed heap for the sequence exactly as it was, making all midconditions equal again. Since $\vec{r}_i = \vec{r}'_i$, EVAL is still applicable.

While sufficient to prove (1), we also notice that it is possible to select a better $\vec{R'}_i$ if some $x \in dom(\vec{P}_i)$ is removed from $\vec{P}_i$ and now exists only in the frame axioms. We denote $\vec{R'}_j$ for all tr_j new frame axioms without any such x. We now apply EVAL to the sequence with the new frame axioms and, since all r_i remain the same, infer $\{\vec{P'}\}t(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r}\}$ where there exists $\vec{R}$ comprising the difference between $\vec{R}_i$ and $\vec{R}'_i$ that proves $\{\vec{P}\}t(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}t(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r}\}$. □

Consistency under $\sqsubseteq_{\vec{\Phi}}$ means that for goal $\mathscr{G}$, if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, then if there exists $\vec{R}_1$ s.t. $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathscr{G}$, then there exists $\vec{R}_2$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2;\vec{r}_2\}$ matches $\mathscr{G}$.

Proof: consistency. Since $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathscr{G}$, we know that $\vec{P}_1 * \vec{R}_1 = \vec{I}$, $\vec{q} \subseteq \vec{Q}_1 * \vec{R}_1$, and $\forall v_i@l_i \in \vec{r}_1.\omega_i = v_i$. Because $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, $\exists\vec{R}.\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1$

$\} \equiv_{\vec{\Phi}} \{\vec{P}_2 * \vec{R}\}p_2\{\vec{Q}_2 * \vec{R};\vec{r}_2\}$, so $\vec{r}_1 = \vec{r}_2$, $\vec{P}_1 = \vec{P}_2 * \vec{R}$, and $\vec{Q}_1 = \vec{Q}_2 * \vec{R}$. From this, $\forall v_i@l_i \in \vec{r}_2.\omega_i = v_i$, $\vec{P}_2 * \vec{R} * \vec{R}_1 = \vec{I}$, and $\vec{q} \subseteq \vec{Q}_2 * \vec{R} * \vec{R}_1$. In other words, there exists $\vec{R}_2 = \vec{R} * \vec{R}_1$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2;\vec{r}_2\}$ matches $\mathscr{G}$. □

4.A.2 Proof of correctness for $(\!|\mathscr{C}|\!)^{reach}$

Theorem 4.A.1 (Theorem 4.5.1)**.** If a valid solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ exists in $(\!|\mathscr{C}|\!)$, then

$$\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}$$

We denote a program that can be part of a valid execution sequence starting at the start states $\vec{I}$ as *reachable*. In our program representation, this means $\vec{I}$ satisfies the preconditions of the sequence. Programs that are not reachable are irrelevant for us, as they cannot participate in any solution. We define a helper predicate for a program $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ being reachable at iteration n as follows:

$$\begin{aligned} reach(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n) \triangleq\ & \exists\vec{R}.\vec{\sigma} * R = \vec{I} \vee \\ & \exists\{\vec{\sigma}_0\}p_0\{\vec{\sigma}_0';\vec{r}_0\},\ldots,\{\vec{\sigma}_i\}p_i\{\vec{\sigma}_i';\vec{r}_i\} \in (\!|\mathscr{C}|\!)_{(n-1)}. \\ & \exists\vec{R}_0,\ldots,\vec{R}_{i+1}.\forall 0 < j \le i.(\vec{\sigma}_{j-1}' * \vec{R_{j-1}}) = (\vec{\sigma}_j * \vec{R}_j) \wedge \\ & \vec{\sigma}_i' * \vec{R}_i = \vec{\sigma} * \vec{R}_{i+1} \wedge \\ & \exists\vec{R}'.\vec{\sigma}_0 * \vec{R}_0 * \vec{R}' = \vec{I} \end{aligned}$$

Proof. We show by induction over n that all programs in $(\!|\mathscr{C}|\!)_{(n)}$ whose precondition holds for $\vec{I}$, or have some precondition $\vec{\sigma}$ and can form a valid execution trace starting a precondition that holds for $\vec{I}$, will also be in $(\!|\mathscr{C}|\!)^{reach}_{(n')}$ for some n'. I.e.,

$$\forall\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)_{(n)}.reach(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n) \implies \exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$$

This implies that for any solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)_{(n)}$, $\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$ as well, which is what $Theorem$ 4.5.1 states.

For $\underline{n = 0}$: For all $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)_{(0)}$, $p = t \in \mathscr{C}$. Since there is no $(\!|\mathscr{C}|\!)_{(-1)}$, the second disjunct of *reach* is irrelevant. That leaves $\exists\vec{R}.\vec{\sigma} * R = \vec{I}$, where by definition $\{\vec{\sigma}\}t\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(0)}$: $\vec{\sigma} = \vec{\sigma}'$ is either $\vec{emp}$ or the initial values of variable t.

For $\underline{n \geq 1}$: We begin by assuming the IH:

$$\forall\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)_{(n-1)}.reach(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}, n-1) \implies \exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$$

Then for all $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)_{(n)}$ where $reach(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}, n)$, there are two cases:

1. If $p = t$ (i.e., $arity(t) = 0$):

   i. if $\vec{\sigma} = \vec{emp}$ or $\vec{\sigma} = \langle x \mapsto \iota_i(x)\rangle_i$, then $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(0)}$.

   ii. for other $\vec{\sigma}$, but $reach(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}, n)$, then from *reach* there exist a sequence of triples $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)_{(n-1)}$ and matching frame axioms $\vec{R}_j$ s.t. $\vec{\sigma}_i * \vec{R}_i = \vec{\sigma} * R_{i+1}$, $\exists\vec{R}'.\vec{\sigma}_0 * \vec{R}_0 * \vec{R}' = \vec{I}$, and $\vec{\sigma}' = \vec{\sigma}$ since t has no effect.

   By the IH, there exists an n' where all $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)^{reach}n'$.

   Since t is not at its original values, it is modified by some some $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$, so by Def. 9, $\{\langle t \mapsto \sigma_j'(t)\rangle_i\}t\{\langle t \mapsto \sigma_j'(t)\rangle_i;\vec{r}'\}$ was added to $(\!|\mathscr{C}|\!)^{reach}_{(n'+1)}$.

   In all cases, $\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$.

2. If p is of the form $t(p_1, \ldots, p_k)$ (i.e. $arity(t) = k > 0$), then its children $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)_{(n-1)}, 1 \leq j \leq k$ and $\exists.\vec{R}_1, \ldots, \vec{R}_k$ s.t. EVAL can be applied to them. $\vec{\sigma} = \vec{\sigma}_1 * \vec{R}_1$.

   Since we know there is a frame axiom $\vec{R}_{i+1}$ s.t. $\vec{\sigma} * \vec{R}_{i+1} = \vec{\sigma}_i * R_i$, this means that the same one can be used for the sequence of arguments to t. This implies that for each $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\}$, $reach(\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\}, n-1)$.

   And so, by the induction hypothesis, $\exists n_j.\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)^{reach}_{(n_j)}$, and if we let $n'' = \max(n_j)$, by the definition of $(\!|\mathscr{C}|\!)^{reach}_{(n)}$, $\forall 1 \leq j \leq k.\{\vec{\sigma}_j\}p_j\{\vec{\sigma}_j';\vec{r}_j\} \in (\!|\mathscr{C}|\!)^{reach}_{(n'')}$. Since all the child sub-expressions of p exist in $(\!|\mathscr{C}|\!)^{reach}_{(n'')}$, p will be enumerated in $(\!|\mathscr{C}|\!)^{reach}_{(n''+1)}$, and so if we let $n' = n''+1$, $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in (\!|\mathscr{C}|\!)^{reach}_{(n')}$.

□

4.A.3 Proof of correctness of $(\!|\mathscr{C}|\!)^{S}$

Theorem 4.A.2 (Correctness of the reduced enumeration)**.** Given $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}_1';\vec{r}_1\} \in (\!|\mathscr{C}|\!)^{reach}$ and a goal $\mathscr{G}_{\vec{\Phi}}$, if there exists $\vec{R}_1$ s.t. $\{\vec{\sigma}_1 * \vec{R}_1\}p_1\{\vec{\sigma}_1' * \vec{R}_1;\vec{r}_1\}$ matches $\mathscr{G}_{\vec{\Phi}}$, then there exists $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\} \in (\!|\mathscr{C}|\!)^{S}$ and a $\vec{R}_2$ s.t. $\{\vec{\sigma}_2 * \vec{R}_2\}p_2\{\vec{\sigma}_2' * \vec{R}_2;\vec{r}_2\}$ also matches $\mathscr{G}_{\vec{\Phi}}$.

First, we show programs are not lost, and equivalent programs in the space still exist:

Lemma 4.A.3. If there exists a triple $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}_1';\vec{r}_1\} \in (\!|\mathscr{C}|\!)^{reach}_{(n)}$ then there is a triple

$$\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\} \in (\!|\mathscr{C}|\!)^{S}_{(n)}$$

s.t. $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}_1';\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$

In other words, the reduced space has an *interchangeable* program under $\sqsubseteq_{\vec{\Phi}}$.

Proof. $\underline{n=0}$: this is true by the definition of $(\!|\mathscr{C}|\!)^{S}_{(0)}$.

$\underline{n>0}$: We assume

$$\forall\{\vec{\sigma}_1\}p_1\{\vec{\sigma}_1';\vec{r}_1\} \in (\!|\mathscr{C}|\!)^{reach}_{(n-1)}.\exists\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\} \in (\!|\mathscr{C}|\!)^{S}_{(n-1)}.\{\vec{\sigma}_1\}p_1\{\vec{\sigma}_1';\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$$

For each $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ in $(\!|\mathscr{C}|\!)^{reach}_{(n)}$:

1. If $p = t$ (i.e., $arity(t) = 0$):

   i. if $\exists\vec{R}.\vec{\sigma} * \vec{R} = \vec{I}$, then by our base case $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\} \in (\!|\mathscr{C}|\!)^{S}_{(0)}$.

ii. if $\neg\exists\vec{R}.\vec{\sigma}*\vec{R}=\vec{I}$, then by definition of $(\!|\mathscr{C}|\!)^{reach}_{(n)}$,

$$\exists tr_1,\dots tr_k \in (\!|\mathscr{C}|\!)^{reach}_{(n-1)}.tr_i = \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r}_i\}$$

and $\exists\vec{R}_1,\dots,\vec{R}_{k+1}$ and $\vec{\sigma}=\langle(t\mapsto\vec{\sigma}_k*R_k)_i(t)\rangle_i$. By the induction hypothesis, there exist $tr'_1,\dots,tr'_k \in (\!|\mathscr{C}|\!)^S_{(n-1)}$ s.t. $tr_i \sqsubseteq_{\vec{\Phi}} tr'_i$. Since the value of t changed since the initial state, it was changed by (i.e., cannot be in the frame of) some tr_i, so it was also changed by tr'_i. So by definition of $(\!|\mathscr{C}|\!)^S_{(n)}$, the triple $\{\vec{\sigma}\}t\{\vec{\sigma};\vec{r}\}$ will be enumerated in $(\!|\mathscr{C}|\!)^S_{(n)}$ at the latest.

2. If $p=t(p_1,\dots,p_k)$: there exist $tr_1,\dots,tr_k\in(\!|\mathscr{C}|\!)^{reach}_{(n-1)}, tr_i=\{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r}_i\}$ s.t. $\exists\vec{R}_1,\dots,\vec{R}_k$ to enable EVAL, and $\vec{\sigma}_1*\vec{R}_1=\vec{\sigma}$. so by the IH there exist $tr'_1,\dots,tr'_k\in(\!|\mathscr{C}|\!)^S_{(n-1)}, tr'_i=\{\vec{\sigma}''_i\}p'_i\{\vec{\sigma}'''_i;\vec{r}'\}$ s.t. $r_i=r'_i$, and for each tr'_i there exists $\vec{R}'_i$ that is the frame axiom for $tr_i\sqsubseteq_{\vec{\Phi}} tr'_i$: $\vec{\sigma}''_i*\vec{R}'_i=\vec{\sigma}_i$ and $\vec{\sigma}'''_i*\vec{R}'_i=\vec{\sigma}'_i$. Therefore, we can use $\vec{R}_1*\vec{R}'_1,\dots,\vec{R}_k*\vec{R}'_k$ to apply EVAL on $tr'_1,\dots,tr'_k$. Since $r_i=r'_i$, the result will still be $\vec{r}$, and will have the same effect, i.e., starting at $\vec{P}'=\vec{\sigma}_1*\vec{R}_1*\vec{R}''_1$ where $\vec{R}''_1$ is the minimal necessary frame axiom for eval ($\vec{R}''_1\subseteq\vec{R}'_1$) EVAL will yield postcondition assertion $\vec{Q}\subseteq\vec{\sigma}'$.

By Theorem 4.3.1, $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\sqsubseteq_{\vec{\Phi}}\{\vec{P}\}t(p'_1,\dots,p'_k)\{\vec{Q};\vec{r}\}$, and since all its components are in $(\!|\mathscr{C}|\!)^S_{(n-1)}$ it will be enumerated by $(\!|\mathscr{C}|\!)^S_{(n)}$.

□

We are now ready to prove our theorem:

Proof of Theorem 4.5.2. Given a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in(\!|\mathscr{C}|\!)^{reach}$ s.t. there exists $\vec{R}$ where $\{\vec{\sigma}*\vec{R}\}p\{\vec{\sigma}'*\vec{R};\vec{r}\}$ matches $\mathscr{G}_{\vec{\Phi}}$, we know from Theorem 4.A.3 that there is a $\{\vec{\sigma}''\}p'\{\vec{\sigma}''';\vec{r}'\}\in(\!|\mathscr{C}|\!)^S$ s.t. $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\sqsubseteq\{\vec{\sigma}''\}p'\{\vec{\sigma}''';\vec{r}'\}$. From Theorem 4.3.1 for *consistency*, we get that if $\{\vec{\sigma}*\vec{R}\}p\{\vec{\sigma}'*\vec{R};\vec{r}\}$ matches $\mathscr{G}_{\vec{\Phi}}$, then there exists $\vec{R}'$ s.t. $\{\vec{\sigma}''*\vec{R}'\}p'\{\vec{\sigma}'''*\vec{R}';\vec{r}'\}$ matches $\mathscr{G}_{\vec{\Phi}}$, or in other words, there exists a solution in $(\!|\mathscr{C}|\!)^S$. □

4.B Evaluation Results

4.B.1 The size of the search space

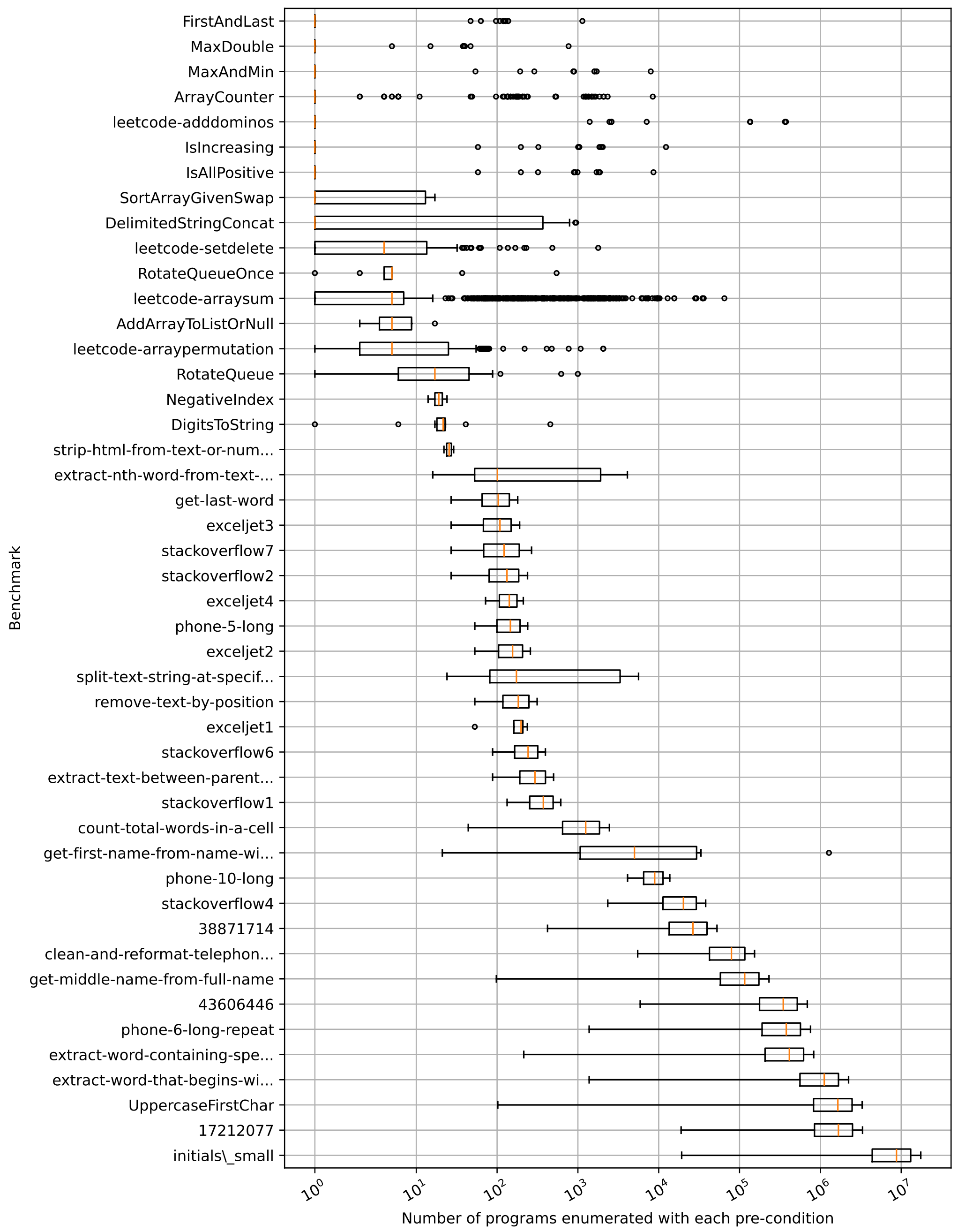


Figure 4.7. The number of programs enumerated with each pre-condition in the bank at termination or timeout.

Chapter 5

LEAP: Live Exploration of AI-Generated Programs

5.1 Introduction

Recent advances in large language models have given rise to AI-powered code suggestion tools like GitHub Copilot [60], Amazon CodeWhisperer [11], and ChatGPT [133]. 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 [13, 129]. 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 Mozannar et al. [129], it is the *single most prevalent activity* when using AI code assistants, and other studies [179, 110, 187, 19] 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 [190, 153, 181, 173]; the former can cause them to abandon the assistant altogether [13], while the latter can introduce bugs and security vulnerabilities [144]. These findings motivate the need for better validation support in

AI-assisted programming environments.

This paper investigates the use of *Live Programming* (LP) [74, 183, 175] as a way to support the validation of AI-generated code. LP environments, such as Projection Boxes [105], 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 [24] and facilitate bug finding [203] and program comprehension [37, 36, 26]. 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 [129], which cause additional cognitive load [187]. 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 [105] 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.

5.2 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 [13, 129, 19]. Moreover, a large-scale survey [110] indicates that 23% of their respondents *have trouble evaluating correctness of generated code*, which echoes the findings of lab studies [179, 13] and a need-finding study [187], where participants report difficulties understanding AI suggestions and express a desire for better validation support. Barke et al. [13] and Liang et al. [110] 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 [173], execution is used less often than quick manual inspection or type checking because it is more time-consuming [13, 110] and interrupts programmers' workflows [187]. The lack of validation support designed for AI-assisted programming, as Wang et al. [187] identify, leads to a higher cognitive load in reviewing suggestions. The high cost of validating AI suggestions, according to some studies [190, 153, 181], 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: Ross et al. [153] investigates a conversational assistant that allows programmers to ask questions about the code, while Vasconcelos et al. [180] 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 [41, 202, 85] and intermediate trace values [141], 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 [52], 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 [74, 183, 175]. Some quantitative studies find that programmers with LP find more bugs [203], fix bugs faster [95], and test a program more often [24].

Others find no effect in knowledge gain [83] or efficiency in code understanding [26]. Still, qualitative evidence points to the helpfulness of LP for program comprehension [37, 36, 26] and debugging [90, 83]. 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 [129, 187].

5.3 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.

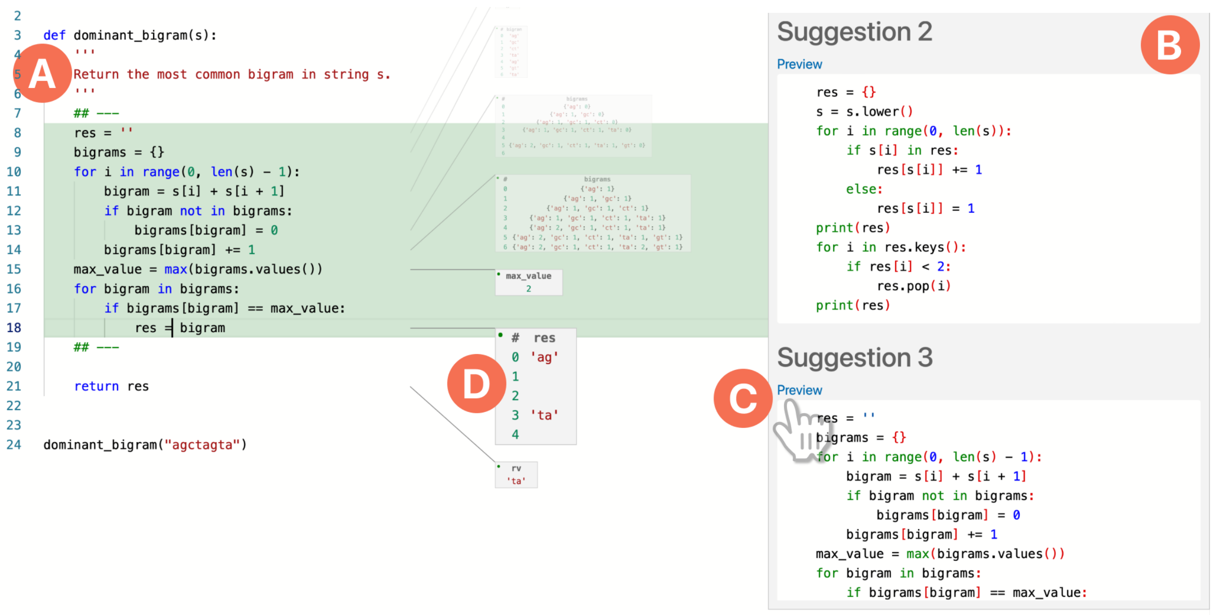
Example Usage

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 fig. 5.1); 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.

- Ⓐ 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.

- Ⓑ 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

[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.



```
2
3   def dominant_bigram(s):
4       '''
5       Return the most common bigram in string s.
6       '''
7       ## ---
8       res = ''
9       bigrams = {}
10      for i in range(0, len(s) - 1):
11          bigram = s[i] + s[i + 1]
12          if bigram not in bigrams:
13              bigrams[bigram] = 0
14          bigrams[bigram] += 1
15      max_value = max(bigrams.values())
16      for bigram in bigrams:
17          if bigrams[bigram] == max_value:
18              res = bigram
19      ## ---
20
21      return res
22
23
24  dominant_bigram("agctagta")
```

```
# bigram
0 'ag'
1 'gc'
2 'ct'
3 'ta'
4 'ag'
5 'gt'
6 'ta'
```

```
#                 bigrams
0                 {'ag': 0}
1            {'ag': 1, 'gc': 0}
2       {'ag': 1, 'gc': 1, 'ct': 0}
3  {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 0}
4
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 0}
6
```

```
#                 bigrams
0                 {'ag': 1}
1            {'ag': 1, 'gc': 1}
2       {'ag': 1, 'gc': 1, 'ct': 1}
3  {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 1}
4  {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1}
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 1}
6 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 2, 'gt': 1}
```

```
max_value
   2
```

```
# res
0 'ag'
1
2
3 'ta'
4
```

```
rv
'ta'
```

Suggestion 2

```
res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
print(res)
for i in res.keys():
    if res[i] < 2:
        res.pop(i)
print(res)
```

Suggestion 3

```
res = ''
bigrams = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram not in bigrams:
        bigrams[bigram] = 0
    bigrams[bigram] += 1
max_value = max(bigrams.values())
for bigram in bigrams:
    if bigrams[bigram] == max_value:
```

Figure 5.1. LEAP is a Python environment that enables validating AI-generated code suggestions via Live Programming.
Ⓐ Users prompt the AI assistant via comments and/or code context. Ⓑ The Suggestion Panel shows the AI-generated suggestions. Ⓒ Pressing a `Preview` button inserts the suggestion into the editor. Ⓓ Users can inspect the runtime behavior of the suggestion in Projection Boxes [105], which are updated continuously as the user edits the code.

then iterates through the dictionary to pick a bigram with the maximum count.

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

- Ⓓ As soon as the suggestion is inserted, Projection Boxes [105] 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 [134], the largest publicly available code-generating model at the time of our study. To support live display of runtime values (fig. 5.1Ⓓ), we built LEAP on top of PROJECTION BOXES, a state-of-the-art LP environment for Python [105] 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.

5.4 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 [60] 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 5.3); (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 fig. 5.1; “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 [43] 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-structur-ed 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 (fig. 5.1Ⓑ) was in focus—and participants' cognitive load (via five NASA Task Load Index (TLX) questions [75]). 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 [28] after the study. Through reflexive interpretation, we used category analysis [197] to assemble the qualitative data into groups. We then revisited notes and recordings to iteratively construct high-level categories.

5.5 Results

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

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

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

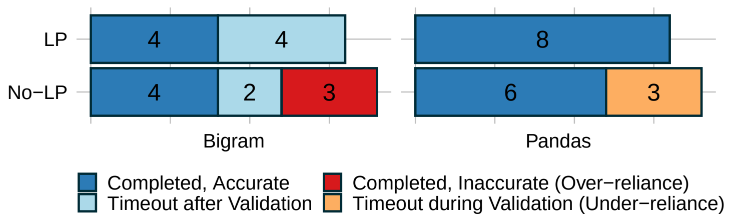


Figure 5.2. Success in validating AI suggestions across groups for Fixed-Prompt tasks. "Completed" means the participant submitted a solution they were satisfied with by the time limit, and "Timeout" means they did not. We deem the validation *successful* if a participant either submitted a solution that was correct (dark blue) or timed out when attempting to fix the correctly identified bugs in their chosen suggestion (light blue).

not discuss them below.[3]

We found six instances of unsuccessful validation, all from the No-LP group. As described in section 6.5, 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 fig. 5.2 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.

[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" ($p = .201, U = 22.5$).

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 5.3) 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 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.

5.5.2 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 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

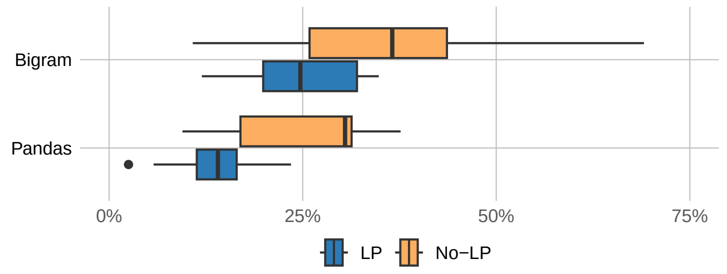


Figure 5.3. Percentage of time spent in the Suggestion Panel across the two groups for Fixed-Prompt tasks.

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; fig. 5.3 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 fig. 5.3, both groups spent more time examining the code in Bigram, while in Pandas they jumped to execution more immediately ($median_{\mathsf{Pandas}} = 16.96\%, median_{\mathsf{Bigram}} = 31.67\%, p = .04, U = 206$). 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."*

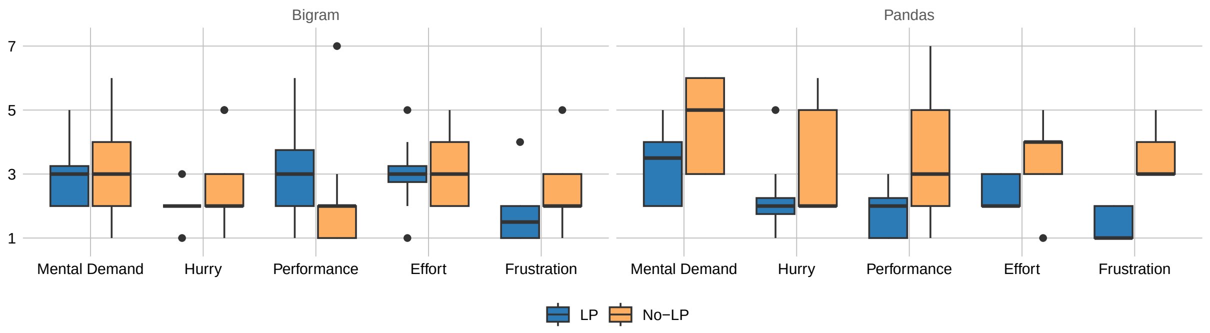


Figure 5.4. NASA Task Load Index (TLX) results for the Fixed-Prompt tasks: Bigram on the left, and Pandas on the right. Higher scores indicate higher cognitive load (in case of Performance this means higher failure rate).

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 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 5.3 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."*

5.5.3 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 [75]: 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."*

5.6 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 (section 5.5.2). Our findings are consistent with prior work [13, 110], 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 5.5.1, 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 [181] 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 5.5.2 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 5.5.2). (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 5.5.3). 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 [187].

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 5.5.2, 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.*

5.7 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 [175, 105] and its need for small demonstrative inputs [166]. 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 [169] for incomplete programs, generating test cases [96, 192], 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 [192]. 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 [52], and further validate the repair in LP.

5.8 Acknowledgements

Chapter 5, in full, is a reprint of Validating AI-Generated Code with Live Programming, as it appears in the Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24). It was done in collaboration with Ruanqianqian (Lisa) Huang, Michael B. James, Sorin Lerner and Nadia Polikarpova. The dissertation author was a primary investigator and co-author of this material.

Chapter 6

ColDeco: An End User Spreadsheet Inspection Tool for AI-Generated Code

6.1 Introduction

In the past two years, large language models (LLMs) [30, 31, 109, 200] have emerged as a practical tool for synthesizing code from natural language. Their commercialization in assistive features such as GitHub Copilot [60, 156] is transforming programming for professional programmers. Still, these tools rely on the expertise of professional programmers to evaluate the (often incorrect) output of the model. The promised value in these tools is only realised through the interactive evaluation and repair of such generated fragments by an expert programmer. However, due to their potential for empowerment, the question arises: how do we design tools specifically to help less skilled users understand and debug AI-generated programs?

6.1.1 Background: AI for end-user programming

Without a formal education in programming, most individuals are unfamiliar with and have difficulties with the abstract concepts such as variables, functions, parameters, etc., that make up the *notional machine* with which a programmer understands how a program functions [42]. Spreadsheets provide mechanisms for individuals unfamiliar with these concepts to have a direct and concrete understanding of their data and transformations on it (such as sums, etc.) due, among other things, to their simplified models of control flow and naming [106, 162, 92, 157]. Spreadsheets can provide a graduated experience that allows a range of individuals, from non-programmers to professional programmers, to solve problems using the tools with which they are most comfortable [130, 40]. Our research investigates the application of AI in synthesizing code solutions for *end-user programmers*, people like the many spreadsheet users who need to code as part of their work but who are not professional programmers [94].

There have been many successful attempts to empower spreadsheet users to define computations without having to learn a formal programming language. The most widely deployed example is FLASHFILL, a programming-by-example string transformation feature that ships commercially [64]. More recently, commercial spreadsheets have introduced AI-powered features for data analytics by synthesizing pivot tables or charts via automatic recommenders or natural language queries [39, 59]. These tools give rise to a new challenge: *How do we enable end users to evaluate the correctness of machine-generated computations without inspecting the underlying code?*

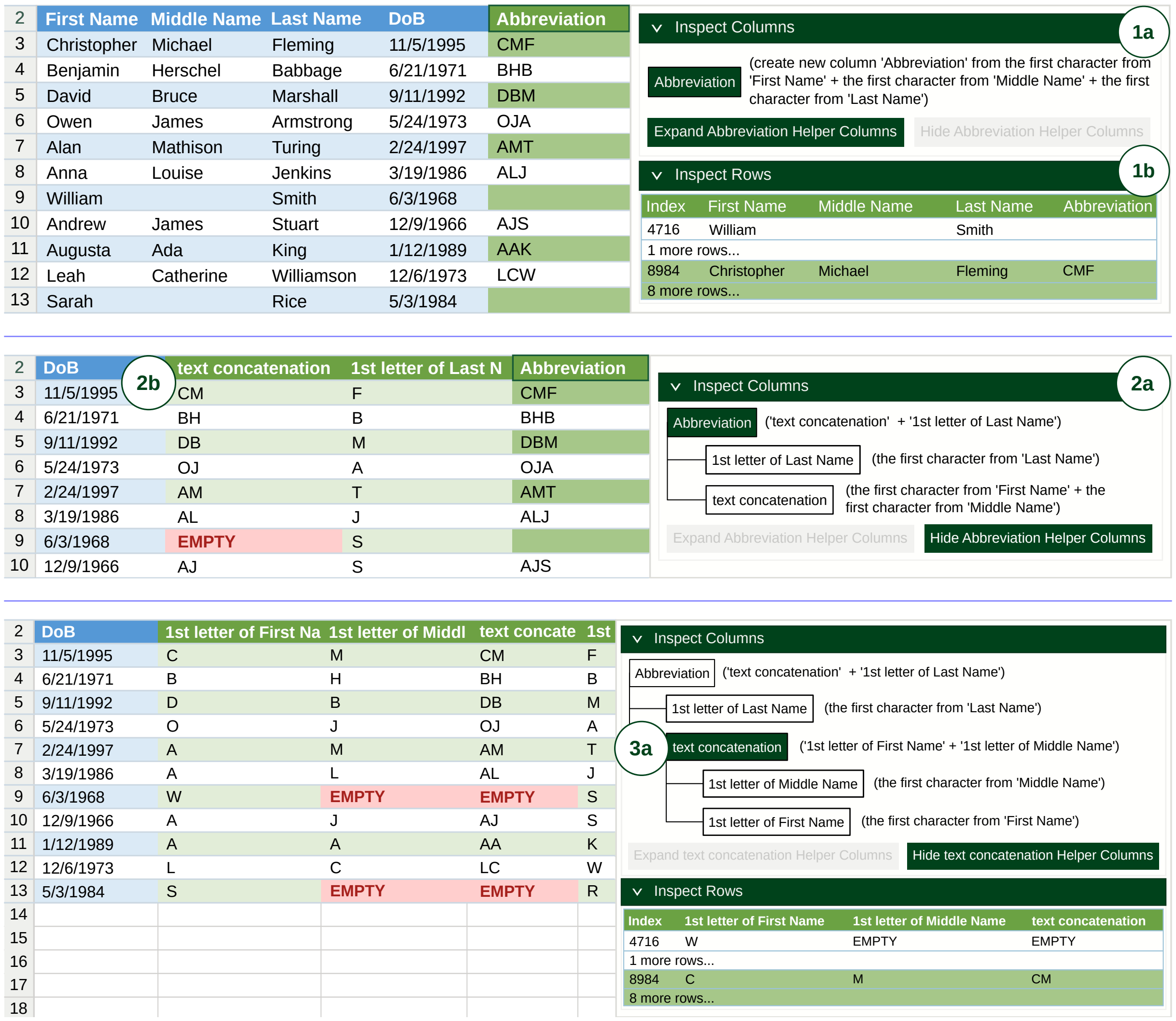
The only recent work we are aware of that targets this challenge for AI-powered spreadsheet programming is *grounded abstraction matching* (GAM), a new interaction style, which explains AI-generated code to end users in natural language [112]. While GAM helps the user confirm that the model's understanding of the problem matches their intent, it does not necessarily help them discover and diagnose errors, when the intent leads to unexpected behavior on the given data.

6.1.2 This paper: COLDECO, an inspection tool for end users

To assist end-user programmers with discovering and diagnosing errors in AI-generated code, COLDECO augments natural-language descriptions in the style of GAM with two complementary features, illustrated in Fig. 6.1.

First, users can *decompose* a generated solution into intermediate helper columns [27] to understand how the problem is solved step by step (see Sec. 6.2.4). Creating helper columns is a common spreadsheet practice for manually simplifying complex formulas. By automating helper column creation, we encourage users to explore the concrete impact of different parts of the code on their data.

Second, users can view a filtered table of *summary rows*, which highlights interesting cases in the program (see Sec. 6.2.3). We introduce an analysis that captures all the unique behaviors that the code displays on the user's data, allowing them to quickly understand the effect of different paths through the code.



2	First Name	Middle Name	Last Name	DoB	Abbreviation
3	Christopher	Michael	Fleming	11/5/1995	CMF
4	Benjamin	Herschel	Babbage	6/21/1971	BHB
5	David	Bruce	Marshall	9/11/1992	DBM
6	Owen	James	Armstrong	5/24/1973	OJA
7	Alan	Mathison	Turing	2/24/1997	AMT
8	Anna	Louise	Jenkins	3/19/1986	ALJ
9	William		Smith	6/3/1968	
10	Andrew	James	Stuart	12/9/1966	AJS
11	Augusta	Ada	King	1/12/1989	AAK
12	Leah	Catherine	Williamson	12/6/1973	LCW
13	Sarah		Rice	5/3/1984	

Inspect Columns 1a

Abbreviation (create new column 'Abbreviation' from the first character from 'First Name' + the first character from 'Middle Name' + the first character from 'Last Name')

Expand Abbreviation Helper Columns | Hide Abbreviation Helper Columns

Inspect Rows 1b

Index	First Name	Middle Name	Last Name	Abbreviation
4716	William		Smith	
1 more rows...				
8984	Christopher	Michael	Fleming	CMF
8 more rows...				

2	DoB	text concatenation	1st letter of Last N	Abbreviation
3	11/5/1995	CM	F	CMF
4	6/21/1971	BH	B	BHB
5	9/11/1992	DB	M	DBM
6	5/24/1973	OJ	A	OJA
7	2/24/1997	AM	T	AMT
8	3/19/1986	AL	J	ALJ
9	6/3/1968	EMPTY	S	
10	12/9/1966	AJ	S	AJS

2b

Inspect Columns 2a

Abbreviation ('text concatenation' + '1st letter of Last Name')

1st letter of Last Name (the first character from 'Last Name')

text concatenation (the first character from 'First Name' + the first character from 'Middle Name')

Expand Abbreviation Helper Columns | Hide Abbreviation Helper Columns

2	DoB	1st letter of First Na	1st letter of Middl	text concate	1st
3	11/5/1995	C	M	CM	F
4	6/21/1971	B	H	BH	B
5	9/11/1992	D	B	DB	M
6	5/24/1973	O	J	OJ	A
7	2/24/1997	A	M	AM	T
8	3/19/1986	A	L	AL	J
9	6/3/1968	W	EMPTY	EMPTY	S
10	12/9/1966	A	J	AJ	S
11	1/12/1989	A	A	AA	K
12	12/6/1973	L	C	LC	W
13	5/3/1984	S	EMPTY	EMPTY	R
14					
15					
16					
17					
18					

Inspect Columns

Abbreviation ('text concatenation' + '1st letter of Last Name')

1st letter of Last Name (the first character from 'Last Name')

3a text concatenation ('1st letter of First Name' + '1st letter of Middle Name')

1st letter of Middle Name (the first character from 'Middle Name')

1st letter of First Name (the first character from 'First Name')

Expand text concatenation Helper Columns | Hide text concatenation Helper Columns

Inspect Rows

Index	1st letter of First Name	1st letter of Middle Name	text concatenation
4716	W	EMPTY	EMPTY
1 more rows...			
8984	C	M	CM
8 more rows...			

Figure 6.1. Initial view (top). Decomposing "Abbreviation" (middle). Decomposing "text concatenation" (bottom).

In general, a debugging experience involves both finding the source of the error (diagnosis) and fixing the error (repair). In ColDeco, we apply the familiar concepts of helper columns, grouping, and filtering in spreadsheets to assist the diagnostic aspect of debugging; we leave considerations of repair for future work (see Sec. 6.7).

Our paper makes the following contributions:

- We present ColDeco, the first end-user inspection tool for comprehending code produced by LLMs for tabular data tasks. ColDeco is an Excel add-in that provides three interrelated interactive components: (1) a decomposition of the solution into intermediate values which are added back to the table in the form of helper columns, (2) summary rows highlighting distinct behaviors in the solution, and (3) a description of the solution expressed in natural language.

- We evaluate this approach with a user study (n = 24) of spreadsheet users of varying expertise completing several debugging tasks. We find that users were able to inspect the results of code generated by an LLM and to find faults when this code was incorrect. Participants regard all three components as useful. Spreadsheet users also indicated that COLDECO may provide value in collaboration by affording users the ability to explain code solutions to colleagues, and better understand complex formulas themselves. We received design feedback for future intelligent user interfaces for end-user debugging of LLMs.

6.2 COLDECO, by example

A data analyst named Kim is cleaning up a spreadsheet of responses to an online form. As a part of their task, they want to create a column of people's initials to use as a part of a unique identifier. Kim has experience with Excel, including using some formulas, but they are not very comfortable with string manipulation functions. So, rather than write a formula themself, they decide to use an LLM to generate the solution and verify it using COLDECO.

6.2.1 Generating the solution

Kim is using a spreadsheet environment that integrates a query box to create new colum-ns using natural language. Kim writes their instruction: *Create a column "Abbreviation" concatenating the first character of each part of the name*, and the spreadsheet queries the LLM to generate code that will compute a new calculated column. The calculated column is added to the spreadsheet and COLDECO is automatically opened as a side pane (Fig. 6.1, top). By convention, additional calculated columns are formatted in green. A cursory glance at the first few rows in the table seems to confirm that the output is reasonable, but Kim would like to make sure the solution is correct, so they turn to COLDECO's inspection features to understand how the output was generated.

6.2.2 A first inspection

The ColDeco pane has two views, *Inspect Columns* (1a) and *Inspect Rows* (1b). The first view provides information about the calculated columns (here: "Abbreviation"), including a description of the calculation generated using natural language templates (1a). The description reveals that the calculation is taking the first letter from each of the "First Name", "Middle Name", and "Last Name" columns, and appears sensible to Kim.

The *Inspect Rows* view clusters the rows from the input table by their behavior in the calculation and depicts one example—a *summary row*—from each cluster (1b). Kim notices there are two clusters, and while the second cluster, with nine rows, behaves correctly, the first cluster, with two rows, is missing the output in the "Abbreviation" column. What initially appeared to be a correct solution is not producing the right output on some of the rows.

6.2.3 Analyzing the summary rows

Kim makes a further inspection of the summary rows to understand the problem. The *Inspect Rows* view (1b) only depicts a subset of the columns from the original table, including the output column ("Abbreviation") and the columns referenced by the calculation ("First Name", "Middle Name", and "Last Name"). Since this view only shows the columns that affect the output, Kim can more easily see that the empty output seems to be related to the empty middle names.

6.2.4 Inspecting the helper columns

To confirm their hypothesis, Kim looks back at the *Inspect Columns* view, and clicks *Expand Abbreviation Helper Columns* to break down the output into its helper columns (Fig. 6.1, middle).

This brings about the *tree view* (2a), which visualizes the structure of the computation. The tree view shows that the output is comprised of two helper columns, one combining the first letter of "First Name" and "Middle Name", and the other just getting the first letter of "Last Name". The names of helper columns are generated using natural language templates similarly to column descriptions. By looking at the values in the table (2b), Kim sees that the "1st letter of Last Name" column is correct for all rows, but for those without a middle name, the "text concatenation" column is showing a red `EMPTY`, indicating that something went wrong there. ColDeco automatically highlights cells that contain errors, where `EMPTY` is analogous to Excel's `#VALUE!`.

Investigating further, Kim selects the "text concatenation" column, and clicks *Expand* once more (Fig. 6.1, bottom). This creates two new helper columns, for the first letter of the first and middle names respectively. Both the *Inspect Columns* and *Inspect Rows* views are synchronized with the grid, such that selecting a column in one of them will update the other views. Kim selects the "text concatenation" column to update the summary rows (3a).

With the intermediate values visible, Kim sees that indeed the program correctly computes the first letter of the first and last names. But for rows without a middle name, computing the first letter fails, and that error propagates, causing the output to be empty as well.

6.3 Related Work

To the best of our knowledge, ColDeco is the first end-user inspection tool for debugging LLM-generated code for data-centric tasks.

The only other work we are aware of with the goal of helping spreadsheet users harness the power of LLMs is *grounded abstraction matching* (GAM) [112]. GAM generates natural-language descriptions of LLM-generated programs, which enable end users to both confirm the model's understanding of their request and to iteratively refine the request if needed. While GAM and ColDeco have a similar target audience, and in fact, our tool incorporates natural-language descriptions similar to those in GAM, the main focus of ColDeco is on two novel mechanisms—helper columns and row summaries—that enable the user to concretely see the effect of the generated code on their data. Apart from GAM, the two most relevant lines of work are those on (1) inspecting synthesized code, and (2) debugging tools for spreadsheets.

6.3.1 Inspecting LLM-Generated Code

There is a growing body of work studying how programmers interact with LLM-power-ed coding assistants [190, 156, 179, 13, 204, 129, 124, 110], and, in particular, how they evaluate and debug LLM-generated code. These studies show that programmers spend a significant proportion of their time inspecting generated code [13, 129] and often have trouble understanding [179] and debugging [13] such code, or evaluating its correctness [110]. Moreover, both programmers [13, 110] and data scientists [124] use multiple sources of information when evaluating AI-generated code, including inspecting the code itself as well as executing the code and inspecting its behavior on concrete inputs. While these studies do not focus on end users, they generally motivate the need for better tool support for understanding and debugging LLM-generated code; in the context of end-user programming there is an additional challenge that the user has no option to inspect the code.

6.3.2 Code Inspection in Program Synthesis

Research on more traditional (search-based) program synthesis has explored multiple ways to help users inspect and disambiguate generated programs, for example, by displaying intermediate values [141] or generating informative examples [202, 85]; unlike COLDECO, these tools assume that the programmer can always fall back on examining the code. A separate line of research focuses on finding relevant data [45, 44, 152] akin to summary rows. These use different techniques, and are complementary to COLDECO.

A more closely related work to ours is FLASHPROG [121], which introduces user interactions for disambiguating multiple synthesized solutions for end-user tasks. Like COLDECO, FLASHPROG aims to improve user confidence in the synthesis result without seeing the code, but it is closely tied to the underlying synthesis algorithm and does not support decomposing solutions.

6.3.3 Debugging Tools for Spreadsheets

The two core features of COLDECO, helper columns and summary rows, are inspired by previous work in end-user programming and spreadsheet research. Automatically created helper columns have been used before to debug user-written formulas [158] or inspect the behavior of code "foraged" from the web [97]. The main difference between these and COLDECO is our interaction model, where the user interactively decomposes the program via the tree view. Our summary rows take inspiration from templates [4], LISH [73], gradual structuring [125], object spreadsheets [122], and calculation view [155]. Each of these addresses the challenge of comprehending and manipulating a large dataset by abstracting it into a smaller structure, which can be a single exemplar row or formula. COLDECO finds a new application for these ideas—evaluating and debugging AI-generated code.

6.4 Design and Implementation

The primary design concept behind ColDeco is to use the familiar structure of the grid to inspect and debug programs. Expansion in the horizontal direction by decomposing helper columns enables users to understand the particular behaviour of an input. Contraction in the vertical direction through filtering or grouping (constructing summary rows) enables users to understand classes of behaviour across all inputs.

The concepts underpinning ColDeco are general purpose. Whilst in this paper we focus on adding a computed column to a table, the design extends to inspecting any expression-based program. By viewing an expression as a function of its inputs, such as $\mathsf{SQRT}(3^2+4^2) \equiv (\lambda(a,b).\mathsf{SQRT}(a^2+b^2))(3,4)$, we can tabulate the function by creating a column for each parameter and the body. Additional rows correspond to combinations of inputs.

ColDeco can be viewed as embedding existing programming language concepts into the grid using widely adopted table interactions. Expanding to add helper columns is like "stepping-in" using a debugger, and contraction via summary rows is like program slicing and case analysis.

6.4.1 Decomposition and helper columns

The tree view of the computed columns allows the user to further decompose them. Decomposing a column simultaneously updates the table with the additional columns, and updates the tree view. For illustration we present the Python pandas code that is associated with the example in Fig. 6.1. The code produced from the user query is:

```
df["Abbreviation"] = df["First Name"].str[0] \
  + df["Middle Name"].str[0] \
  + df["Last Name"].str[0]
```

When the expression associated with a computed columns contains no sub-redexes, such as `df["First Name"].str[0]`, the expand button in the view is disabled. In our design we also considered other modalities for decomposing columns, such as context menus triggered from the grid, but limitations in the prototyping framework made this difficult. Consequently, we focused only on the inspection view.

When a user decomposes a column we extract the subexpressions and assign them as new columns to the data frame, which is then recomputed. For example, decomposing the `"Abbreviation"` column produces the following code that corresponds to Fig. 6.1 (middle). Column names are abbreviated using (...).

```
df["text..."] = df["First..."].str[0] + df["Middle..."].str[0]
df["1st letter..."] = df["Last..."].str[0]
df["Abbr..."] = df["text..."] + df["1st letter..."]
```

6.4.2 Summary rows

When an input table is large it can be difficult to verify that a generated solution satisfies the requirements. Summary rows are an automatic technique to highlight these significant inputs for end-users, who would otherwise be unaware of such programming techniques [78].

The core idea behind the implementation of summary rows is that we group rows by the composite behaviors of their columns, where *behavior* is defined by a partitioning of values. We chose this design because it naturally extends onto the concept of a table, with the intent that table partitioning provides an approachable way to understand program case analysis. As we discuss in Sec. 6.6.5, some users still faced challenges understanding row summaries but identified extensions that would significantly improve comprehension of row summaries. We now describe the implementation in detail.

To compute the summary rows we derive a key for each row. The key is derived by considering the fully decomposed table, regardless of view state, and applying a series of *tagging* predicates to every value in that row; the key is formed from the disjoint union of tags in the row. The tags include attributes such as type, sign, and truthiness. The table in Fig. 6.1 has two unique tag sets, with the distinguishing tag being the type of the "Middle Name" column which is `string` or `EMPTY`[1].

[1]The columns derived from the middle name that are also subsequently empty, such as the first letter, also contribute distinguishing tags. We omit the full list for brevity.

6.5 User Study

We conducted an hour-long within-subjects study with 24 participants to answer the following research questions:

(RQ1) Do COLDECO's features enable users to correctly diagnose generated code outputs in spreadsheets?

(RQ2) How does decomposing the output into helper columns affect users' ability to diagnose the output?

(RQ3) What are users' perception of the usefulness of each of the features for inspecting COLDECO's output?

6.5.1 Participants

For this study, we recruited 24 participants, 10 women and 14 men, across 12 professions. 19 participants reported having "a lot of experience" with spreadsheet software, but all had at least some experience. All participants also had some experience writing spreadsheet formulas, with 8 using "a variety of different functions" and the others only using "a few basic functions such as `SUM` and `AVERAGE`". For traditional programming languages, 18 participants were at least "moderately experienced" programmers, with the others knowing only enough for "small infrequent tasks" or with little to no experience.

6.5.2 Tasks

Each participant was asked to solve 4 tasks inspired by the WREX study [41], and Excel questions on StackOverflow[2]. Each task includes a table of data[3], a task description, and a pre-written query for COLDECO[4]:

A1) Given a table of purchase data, create a column containing the total amount paid after discounts and reimbursements for each entry.

[2]https://stackoverflow.com/questions/tagged/excel-formula
[3]Taken from [41, 138, 148] or created for the study.
[4]You can find the the study material in the technical report [50].

A2) Given a table of TV shows, create a column containing "Yes" if a show's popularity is $\geq 1,000$ or it has a vote average of ≥ 8.0 with at least 10,000 votes.

B1) Given a table of event dates and locations, create a column containing the duration of each event in hours.

B2) Given a table of books, create a column which rounds the price of each book such that the last digit of the rounded price is the nearest 4, 5, or 9.

The query given for A1 and B1 resulted in a correct solution, while A2 and B2 had bugs affecting a small set of the rows. We grouped the tasks into two pairs (A1, A2) and (B1, B2) of approximately the same difficulty (A1 and B1 are simpler and A2 and B2 are more complex).

For each task, participants were asked to use a pre-written query to generate an output, and use COLDECO's inspection features to *diagnose* if the output matches the task's description. To finish the task, they were asked if the output is correct (their "diagnosis"), their confidence in their diagnosis, and (if they diagnosed it as incorrect) what query they would try next to get a correct output.

Since the focus of the study is validating COLDECO's output, not completing tasks, we controlled for the variability in input queries by providing users with the query to use, and did not ask them to try to get to a correct output. Asking them about the query they would try next enabled us to confirm that participants diagnosed the correct cause for incorrect outputs.

6.5.3 Study protocol

We study two configurations of COLDECO, `HC` and `No-HC`, which differ in the availability of the *helper columns* feature (the `No-HC` version does not have the *Expand* button in the *Inspect Columns* view). We chose to isolate the effects of the helper columns in particular, since COLDECO has multiple interacting features and column decomposition is the most novel aspect. We did not compare against a traditional "control" condition, since we were not aware of any comparable tools, and we believe that access to COLDECO's features would likely be trivially better than having access to no debugging tools at all.

We had four randomly-assigned groups of 6 participants, based on the condition they were assigned to first (`HC`-first vs. `No-HC`-first) and the task pair (A-first vs. B-first).

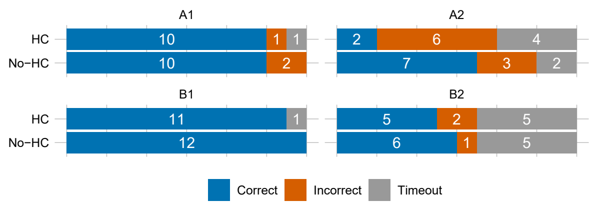


Figure 6.2. The number of participants whose diagnosis for each was correct, incorrect, or they ran out of time.

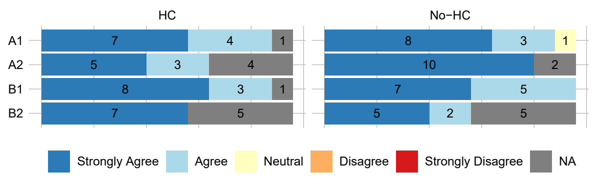


Figure 6.3. Participants' agreement with the sentence "I am confidence about my answer".

The study was conducted remotely via video conferencing, with participants controlling COLD-ECO on the investigator's machine. Each study session began with a tutorial of the tool (using the example covered in Sec. 6.2), followed by a warm-up where users were asked to repeat the tutorial steps and ask any questions. They then performed the first pair of tasks, followed by a mid-study survey, the second pair of tasks and the post-study survey, ending with a semi-structured interview.

Users had at most 7 minutes to perform each task. If any task exceeded that limit, the investigator informed the participant that they were out of time, and moved to the next task or survey. Participants were encouraged to think-aloud throughout the study.

The participants who saw the `No-HC` condition first were given the column decomposition portion of the tutorial after the mid-study survey, while those in `HC` were simply informed that they will not have access to the *Expand* button for the second pair of tasks. Each group only answered questions about column decomposition in the survey immediately following their use of the feature.

6.6 Results

Users completed task A1 with a mean time of 4.37 minutes (SD=1.42), A2 in 4.63 minutes (SD=1.46), B1 in 3.38 minutes (SD=1.40) and B2 in 4.67 minutes (SD=1.23), excluding time-outs. Since this was a think-aloud study, we do not compare times across conditions.

6.6.1 Correctness and confidence

Users across both conditions performed well (Fig. 6.2) and were confident (Fig. 6.3) on the simpler tasks, but they did not perform as well in A2 and B2. We attribute this to time pressure, where users were not able to inspect the outputs with the same level of detail, and it is possible that the additional cognitive load caused by the more complicated tasks impacted users' performance. Given these results **we answer RQ1 with a tentative yes**: without the artificial time constraint imposed by the study, users are able to correctly diagnose generated code, though future studies are needed to confirm that this generalizes to longer and more complex tasks.

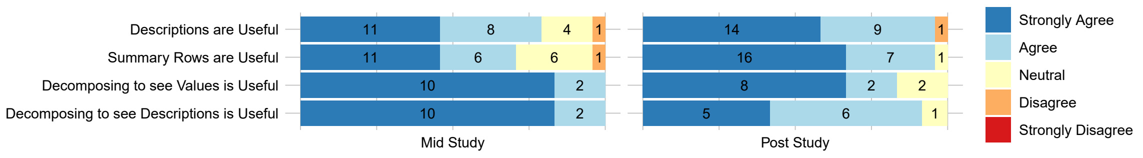


Figure 6.4. The results from the surveys. For each question, participants were asked to select their agreement with the statement on a five-level Likert scale. Note that participants in each condition only answered questions about decomposition in the survey immediately following their use of the feature, so the last 2 rows show 12 responses each.

6.6.2 The effect of column decomposition

Fig. 6.2 shows a lower performance on A2 for the `HC` group—six incorrect diagnoses vs. three for `No-HC`—although this difference was not statistically significant (Fisher's Exact Test). The significantly lower confidence (Wilcoxon Rank Sum Test, $p = 0.046$) for A2 in this group (Fig. 6.3), and the fact that we don't see this effect in B2, suggests that this is not due to decomposition misleading users. Rather, we hypothesize that it is due to the program used in A2: it was a chain of if-else statements, with a bug in the final case. Users needed to decompose the program four times before reaching the values they were interested in. So the complexity of the interaction, combined with relatively little experience with the tool may have overwhelmed and frustrated users.

So **to answer RQ2**, we did not find evidence of significant improvements resulting from the presence of helper columns based only on the quantitative results. We believe, however, this is due to the limitations in our study design (the combination of time pressure and the complexity of the task); indeed, our qualitative evaluation of helper columns, discussed below, is much more positive.

6.6.3 Survey results

Fig. 6.4 presents participants' answers to the mid- and post-study surveys. There was no significant difference between the conditions, so we report the responses to shared questions in aggregate. In the post-study survey, users rated both the natural-language descriptions (M = 4.50, SD = 0.722) and summary rows (M = 4.625, SD = 0.576) highly. We also found a significant *improvement* in the ratings for summary rows between the mid- and post-study surveys (Wilcoxon Signed Rank test, p = 0.016). As we discuss in 6.6.5, this suggests a learning effect for this feature, which we found to be harder to learn than others.

Since participants only had access to column decomposition in half of the study, we included questions about that feature only in the survey immediately following that half (the mid-study survey for `HC`-first, the post-study survey for `No-HC`-first). Despite the lower performance noted above, participants rated decomposition highly, for inspecting both the values (M = 4.5, SD = 0.798) and the descriptions (M = 4.33, SD = 0.651) of helper columns.

6.6.4 Ranking the features

As a part of the post-study interview, we asked participants to rank the three features (see Fig. 6.5). We found that there was a diversity of preferences for the features, with no clear preference shared by all. Two participants, P21 and P23, ranked decomposition and natural-language descriptions in combination as their first choice. P1 similarly ranked them together, but in last place. P4 considered all features equally useful. The summary rows tended to be ranked lower than others.

Given the ranking and survey results, **we can answer RQ3**: participants found all three features useful, and there was no clear preference for any one feature. However, the summary rows feature was ranked lower than others, which we attribute to its steeper learning curve.

6.6.5 Qualitative analysis

We transcribed the semi-structured interviews and participants' comments, and the first author performed an inductive thematic analysis of the transcript (through open-coding) [23]. Here, we discuss notable themes from the analysis, and how they pertain to the results above.

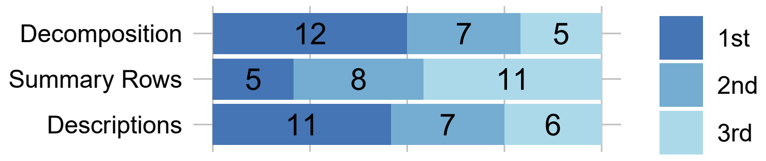


Figure 6.5. The number of participants who ranked each feature in a particular position. Some ranked two or all three features equally.

Decomposition affords better transparency and analysis

Participants were positive about decomposition, and many commented on its utility during the survey and while ranking the features. Several participants stated that decomposition afforded them a better understanding of the steps involved in the output (P7, 14, 17, 19, 23), with P19 referring to it as a "show-your-work button", and P23 saying that decomposition "makes it less [...] like a black box". Others (P15, 19, 21, 22) noted its role in finding the precise cause of an error, saying that it helps "drill the formula down" (P22) and "pinpoint exactly which part of the prompt is not working well" (P15).

ColDeco for collaboration

Participants were excited to use ColDeco for collaboration with their colleagues. P11 and P15 noted that the helper columns would help them *explain* their work to someone else. P6 and P19 commented that natural-language descriptions would help them *understand* complex formulas written by others. And P6 and P15 considered using natural-language descriptions to automatically *document* their spreadsheets.

Difficulties with summary rows

We found that users tended to struggle with forming a usable mental model of the summary rows, instead preferring to manually inspect the values in the grid. P6 mentioned that "I don't really understand it, so I wanted to look at the table myself", and P20 stated "I still don't know what it means." Users' comments indicate that this is not due to the nature of the information presented in the summary rows view, but rather that lack of transparency makes it hard to understand how it works and how it can be useful.

Confirming the learning effect noted in Sec. 6.6.3, some users mentioned that they needed practice to use summary rows effectively. P2 noted that “I feel like once I start using it I might get a grasp of what’s happening there”, and P14 only used it for their last task, saying “I think I [didn’t] understand [summary rows] before this [...] Maybe I got used to it, because it’s my fourth time using this program.”

Others mentioned that more transparency would help their understanding. P4 suggested that “it would be helpful in [summary rows], when it’s showing different categories, to specify what the differences are”, and similarly P5 wanted “a drop-down that would give you a description”. P3, P7 and P21 called for the ability to click on groups to see more example rows for that group, while P21 suggested using a different color to distinguish each behavior.

The usefulness of the information presented in the summary rows view is further confirmed by users who *did* form a usable mental model of the view, and noted its effectiveness in finding errors, especially in combination with other features. P12 found it very helpful “because it brings the different outcomes and behaviors to the front of the screen very quickly.” And P16 and P22 mentioned that it would be the first feature they would use, as it lets them quickly check for multiple or incorrect behaviors.

Decomposition design suggestions

Alongside the broadly positive response, some participants noted certain limitations with the current design of column decomposition and suggested improvements (P4, 12, 17, 19, 24). The main issue was that, with the more complex tasks, the number of times the output could be decomposed grew, resulting in a large number of helper columns and a more complex tree view.

For instance, P24 noted that “As I kept expanding, I kept seeing the other columns for the other cases [...] so it got confusing which ones were related to *just* [...] ones I was actually interested in at that time”. As a solution to this, P19 mentioned wanting the ability to selectively expand subexpressions, imagining a design in which they could “highlight part of the formula that I’m interested in and say ‘Show me this as a helper column” ’.

6.6.6 Threats to validity

The most notable threat to *internal* validity of our study is the time limit for each task, which led to some participants timing out, particularly for the more complex tasks A2 and B2. However, users were quite effective at using the tool despite the time limit, which suggests that given more time, users' effectiveness and perceptions may improve. Another threat to internal validity is Participant Response Bias [38]. Following recommendations from [176], we tried to mitigate inflation in subjective ratings and qualitative feedback by presenting multiple designs within ColDeco to the participants.

Our most significant threat to *external* validity is that a majority of our participants had moderate-to-high programming experience. Our participants nevertheless present an important subset of spreadsheet users, as they have a variety of professions and formula experience. Another threat to external validity is that the tasks and data may not represent Excel's real-world use cases. To address this, we used real questions from StackOverflow to inform our tasks, and used real-world data where applicable.

6.7 Discussion

ColDeco was built to help us understand how to help users diagnose faults in AI-generated code solutions for tabular data problems. Our study identified the following areas for improvement.

Handling complicated tasks. Users found that for a more complex task, the number of helper columns shown could become overwhelming. An improvement would provide a program slicing [189] capability that could prune the helper columns to only show immediately relevant columns to the calculation of a particular value.

Explaining summary rows. Some users did not understand the meaning or purpose of the row summaries. Based on this feedback, improvements include better documentation, generating natural language explanations of the groupings, or including automatically generated insights about core differences between the groupings.

Handling different kinds of input and output data. Our prototype supports diagnosing errors for a limited, but important, subset of Excel tables (single flat column-major tables). Because code-generating AI can produce code solutions for a wider variety of contexts, it is important to consider how the approaches we outline generalize. Similarly we focused on code that generates columns of results but more general outputs, such as single values and new tables, must be considered.

6.8 Conclusion

In the near future, LLMs will be used both by professional programmers and non-professional users to write code. Giving end users confidence and trust in the code that AI systems generate is one of the most important design challenges in empowering millions more to program.

We present COLDECO, a new spreadsheet user experience designed to give users confidence that the code being generated by the LLM is correct. To the best of our knowledge, COLDECO is the first end-user inspection tool for comprehending code produced by LLMs for tabular data tasks. COLDECO provides *summary rows*, which highlight collections of rows that exhibit distinct behaviors in the code, and *helper columns*, which map the results of sub-computations back on the table.

We evaluate COLDECO using a within-subjects user study (n = 24) where participants are asked to verify the correctness of programs generated by a language model. In both quantitative and qualitative measures, our subjects found row summaries and helper columns were valuable in understanding the AI-generated code solutions. We found that while all three features are independently useful, participants preferred them in combination. Users especially noted the usefulness of helper columns and natural language explanations, but wanted more transparency in how summary rows are generated to assist with understanding and trusting them.

Topics for future work include understanding the application of COLDECO in collaborative settings for explaining and understanding existing formulas. This aspect of using COLDECO was highlighted by our users and further investigation is needed. While COLDECO focused on detecting and diagnosing potential errors, a natural and important extension of this capability is the ability to repair errors once they are found. Integrating repair into the COLDECO experience is an important topic for further study.

6.9 Acknowledgements

Chapter 6, in full, is a reprint of COLDECO An End User Spreadsheet Inspection Tool for AI-Generated Code, as it appears in the 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). It was done in collaboration with Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar and Benjamin Zorn. The dissertation author was a primary investigator and author of this paper.

Chapter 7

Conclusion

This dissertation explored the human-centered dimensions of Program Synthesis, looking at both enumerative and LLM-based approaches to generating code. It considered interfaces and interactions for providing specifications and validating the resulting code, and expanded bottom-up synthesis algorithms with support for side-effects and control structures. In this last chapter, I briefly touch on future directions for the work discussed in this dissertation. This is, by nature, speculative, but I hope that the reader will be informed and inspired to explore some of the ideas that, if circumstances had permitted, I would have done myself.

7.1 Future Works

In his 2013 revision of the levels of liveness, aside from the 4 levels in Fig. 1.3, Tanimoto introduced two new levels with an eye towards the future [175]. Named “Tactically” and “Strategically Predictive” respectively, he focused on the ability of some future agent (possibly through the use of machine learning) to assist the programmer by predicting small-scale code edits and additions for level five, and larger-scale changes to the whole software at level six. The recent works on LLM-driven code generation, including LEAP, can be seen as instantiations of level five liveness.

But, as I mentioned in Ch. 1, the focus of this dissertation, as well as a majority of research on human-centered program synthesis, is on synchronous program synthesis of a particular form: the user provides a specification, waits for the synthesizer, and validates the results. However, this is far from the only possible interaction, and with the flexibility that LLMs provide, we have the opportunity to explore many other ways in which similar techniques can assist programmers. One such interaction is conversational [153], using chat interfaces to allow for a more flexible interaction with an LLM. But here I would like to suggest two other promising designs, one for each new level of liveness.

A True Programmer's Assistant. The first is to invert the relationship between the human and the synthesizer. Typical synthesizers generate the code, and leave the arguably more arduous and tedious task of validation to the user. But we can design an inverted interaction where the user drives the programming, and the synthesizer "looks over the shoulder" of the programmer, making helpful suggestions, looking up relevant documentation or library functions, and catching human error. As far as I am aware, no such tools have been developed at time of writing, though the Programmer's Assistant [153] users expressed an interest in such an assistant that is more "proactive", and others [87, 124] have offered discussion towards such a design.

Asynchronous Program Synthesis. The other design is to leverage the flexibility of LLMs for asynchronous synthesis. LLMs are capable of breaking down a given task into intermediate steps [188] and repairing and refining their work [118], both of which can significantly improve their performance. Doing so is too slow and costly for a synchronous model, but in an asynchronous setting where the synthesizer is given one major task and may generate larger amounts of code, or modify multiple parts of a codebase, the cost may be justified. And we could leverage existing interfaces such as issue trackers for specification, and testing and code review to allow for better human validation after the fact, a feature that previous work suggests would also address users' concerns about AI-generated code [190].

Bibliography

[1] 2019. Alfie. https://alfie.prodo.ai/. Accessed: 2019-09-01.

[2] 2019. LightTable. http://lighttable.com/. Accessed: 2019-09-01.

[3] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 270–288.

[4] Robin Abraham, Martin Erwig, Steve Kollmansberger, and Ethan Seifert. 2005. Visual specifications of correct spreadsheets. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 189–196.

[5] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *2021 IEEE Symposium on Visual Languages and Human- Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576170 ISSN: 1943-6106.

[6] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950.

[7] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

[8] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings , Part II*. 163–179.

[9] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. *Electronic Proceedings in Theoretical Computer Science* 260 (Nov. 2017), 97–115. https://doi.org/10.4204/eptcs.260.9

[10] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. 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. 319–336.

[11] Amazon. 2023. CodeWhisperer. https://aws.amazon.com/codewhisperer/.

[12] Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. 2019. Extending enumerative function synthesis via SMT-driven classification. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 212–220.

[13] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[14] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[15] Shaon Barman, Rastislav Bodik, Satish Chandra, Emina Torlak, Arka Bhattacharya, and David Culler. 2015. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 121–136.

[16] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

[17] Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: Program Synthesis by Reasoning over Graphs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 134 (oct 2021), 29 pages. https://doi.org/10.1145/3485511

[18] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. https://doi.org/10.1109/ICSM.2015.7332518

[19] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (Dec. 2022), 35–57. https://doi.org/10.1145/3582083

[20] Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual*. The MIT Press. https://doi.org/10.7551/mitpress/13770.001.0001 arXiv:https://direct.mit.edu/book-pdf/2085914/book_9780262372633.pdf

[21] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. 2013. Proof-Directed Parallelization Synthesis by Separation Logic. *ACM Trans. Program. Lang. Syst.* 35, 2, Article 8 (jul 2013),

60 pages. https://doi.org/10.1145/2491522.2491525

[22] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging , Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[23] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[24] Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. 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*. ACM, Boise ID USA, 227–236. https://doi.org/10.1145/3311927.3323138

[25] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (jan 2023), 30 pages. https://doi.org/10.1145/3571226

[26] Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. 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*. ACM, Amsterdam Netherlands, 8.

[27] George Chalhoub and Advait Sarkar. 2022. "It's Freedom to Put Things Where My Mind Wants"': Understanding and Improving the User Experience of Structuring Data in Spreadsheets. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 585, 24 pages. https://doi.org/10.1145/3491102.3501833

[28] Kathy Charmaz. 2014. *Constructing Grounded Theory*. SAGE.

[29] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

[30] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa,

Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[31] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[32] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

[33] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. Live coding in laptop performance. *Organised Sound* 8, 3 (2003), 321–330. https://doi.org/10.1017/S135577180300030X

[34] João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. 2022. Derivations with Holes for Concept-Based Program Synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 63–79.

[35] Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. 2020. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 141–168.

[36] Robert DeLine and Danyel Fisher. 2015. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human- Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[37] Robert A DeLine. 2021. 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*. ACM, Yokohama Japan, 1–11. https://doi.org/10.1145/3411764.3445267

[38] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is better!": participant response bias in HCI. In *CHI*. ACM, 1321–1330.

[39] Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. 2017. Analyza: Exploring Data with Conversation. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces* (Limassol, Cyprus) *(IUI '17)*. Association for Computing Machinery, New York, NY, USA, 493–504. https://doi.org/10.1145/3025171.3025227

[40] Paul Dourish. 2017. *The stuff of bits: An essay on the materialities of information*. MIT Press, Chapter Spreadsheets and Spreadsheet Events in Organizational Life.

[41] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

[42] Benedict Du Boulay. 1986. Some difficulties of learning to program. *J. Educational Computing Research* 2, 1 (1986).

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

[44] Anna Fariha, Matteo Brucato, Peter J. Haas, and Alexandra Meliou. 2020. SuDocu: Summarizing Documents by Example. *Proc. VLDB Endow.* 13, 12 (aug 2020), 2861–2864. https://doi.org/10.14778/3415478.3415494

[45] Anna Fariha and Alexandra Meliou. 2019. Example-Driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1262–1275. https://doi.org/10.14778/3342263.3342266

[46] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, Crowd-Scale Programming Practice in the IDE. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2491–2500. https://doi.org/10.1145/2556288.2556998

[47] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[48] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[49] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*.

[50] Kasra Ferdowsi, Jack Williams, Ian Drosos, Andy Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar, and Ben Zorn. 2023. *ColDeco: An End User Spreadsheet Inspection Tool for AI -Generated Code*. Technical Report. Microsoft Research. https://www.microsoft.com/en-us/research/publication/coldeco-an-end-user-spreadsheet-inspection-tool-for-ai-generated-code/

[51] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (10 2021). https://doi.org/10.1145/3485530

[52] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

[53] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. 2023. Inductive Program Synthesis Guided by Observational Program Similarity. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 912–940.

[54] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

[55] Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 164 (jun 2023), 24 pages. https://doi.org/10.1145/3591278

[56] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

[57] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/2568225.2568250

[58] Katy Ilonka Gero, Zahra Ashktorab, Casey Dugan, Qian Pan, James Johnson, Werner Geyer, Maria Ruiz, Sarah Miller, David R Millen, Murray Campbell, et al. 2020. Mental Models of AI Agents in a Cooperative Game Setting. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[59] Mar Ginés. 2022. *Better insights from Analyze Data feature in Excel.* Microsoft Corporation. https://insider.office.com/en-us/blog/better-insights-from-analyze-data-feature-in-excel Office Insider Blog.

[60] GitHub. 2023. GitHub Copilot - Your AI pair programmer. https://copilot.github.com/.

[61] Andrew D. Gordon, Carina Negreanu, José Cambronero, Rasika Chakravarthy, Ian Drosos, Hao Fang, Bhaskar Mitra, Hannah Richardson, Advait Sarkar, Stephanie Simmons, Jack Williams, and Ben Zorn. 2023. Co-audit: tools to help humans double-check AI-generated content. (2023). arXiv:2310.01297 [cs.HC]

[62] Susanne Graf and Hassen Saidi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings 9*. Springer, 72–83.

[63] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[64] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[65] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *PLDI*. ACM, 62–73.

[66] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. 2015. StriSynth: synthesis for live programming. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 701–704.

[67] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and trends in programming languages* 10, 1-2 (2017), 1–119.

[68] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[69] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[70] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. 2021. RbSyn: type-and effect-guided program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 344–358.

[71] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1584–1607.

[72] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

[73] Alan Hall, Michel Wermelinger, Tony Hirst, and Santi Phithakkitnukoon. 2018. Structuring Spreadsheets with the "Lish" Data Model. *arXiv preprint arXiv:1801.08603* (2018).

[74] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph. D. Dissertation. Cambridge, MA, USA. AAI0805688.

[75] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–

183.

[76] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2022.16* (2022). Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2022.16

[77] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292.

[78] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. 2016. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 56–65. https://doi.org/10.1109/SANER.2016.86

[79] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 72, 18 pages. https://doi.org/10.1145/3586183.3606733

[80] Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D'Antoni. 2019. Direct Manipulation for Imperative Programs. In *International Static Analysis Symposium*. Springer, 347–367.

[81] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, 162–171.

[82] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

[83] Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. 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 (SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 495–501. https://doi.org/10.1145/3478431.3499305

[84] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

[85] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia

Polikarpova. 2020. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 205:1–205:27. https://doi.org/10.1145/3428273

[86] Michael B. James and Nadia Polikarpova. 2022. Program Recognition in Synthesis. (6 2022). https://doi.org/10.1184/R1/19787572.v1

[87] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. https://doi.org/10.1145/3526113.3545659

[88] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[89] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1143–1158.

[90] Hyeonsu Kang and Philip J. Guo. 2017. 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* (Qu&#233;bec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[91] Saketh Ram Kasibatla. 2018. *Seymour: A Live Programming Environment for the Classroom*. Master's thesis. University of California, Los Angeles.

[92] Alan Kay. 1984. Computer Software. *Scientific American* 251, 3 (1984), 52–59. http://www.jstor.org/stable/24920344

[93] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 407–426.

[94] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44.

[95] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How Live Coding Affects Developers' Coding Behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013 ISSN: 1943-6106.

[96] Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty,

Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive Code Generation via Test-Driven User-Intent Formalization. https://doi.org/10.48550/arXiv.2208.05950 arXiv:2208.05950 [cs]

[97] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting End-User Programmers Who Transmogrify Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 311, 12 pages. https://doi.org/10.1145/3411764.3445265

[98] Tessa Lau et al. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*.

[99] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration.. In *ICML*. 527–534.

[100] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[101] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 376–379.

[102] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[103] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[104] K. Rustan M. Leino and Aleksandar Milicevic. 2012. Program Extrapolation with Jennisys. In *OOPSLA*. ACM, 411–430.

[105] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[106] Clayton Lewis and Gary M Olson. 1987. Can principles of cognition lower the barriers to programming. In *Empirical studies of programmers: second workshop*, Vol. 2. Ablex Publishing Corporation, Norwood, New Jersey, 248–263.

[107] Hongwei Li, Zhao Xuejiao, Zhenchang Xing, Lingfeng Bao, Xin Peng, Dongjing Gao, and Wenyun Zhao. 2015. AmAssist: In-IDE ambient search of online programming resources. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (04 2015), 390–398. https://doi.org/10.1109/SANER.2015.7081849

[108] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient Bottom-Up Synthesis for Programs with Local Variables. *Proceedings of the ACM on Programming Languages* 8, POPL (2024).

[109] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814* (2022).

[110] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2023. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. https://doi.org/10.48550/arXiv.2303.17125 arXiv:2303.17125 [cs.SE]

[111] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. ACM, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[112] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. https://doi.org/10.1145/3544548.3580817

[113] Calvin Loncaric, Michael D Ernst, and Emina Torlak. 2018. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*. 958–968.

[114] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast synthesis of fast collections. In *PLDI 2016: Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation*. Santa Barbara, CA, USA, 355–368.

[115] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[116] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices* 51, 1 (2016), 298–312.

[117] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29.

[118] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri

Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL]

[119] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

[120] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276497

[121] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 291–301. https://doi.org/10.1145/2807442.2807459

[122] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. Object spreadsheets: A new computational model for end-user development of data-centric web applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 112–127.

[123] Gayle Laakmann McDowell. 2015. *Cracking the coding interview: 189 programming questions and solutions; 6th ed.* CareerCup, Palo Alto, CA. https://cds.cern.ch/record/2669252

[124] Andrew M. McNutt, Chenglong Wang, Robert A. DeLine, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. arXiv:2301.11178 [cs.HC]

[125] Gary Miller and Felienne Hermans. 2016. Gradual structuring in the spreadsheet paradigm. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 240–241.

[126] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[127] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

[128] Ashish Mishra and Suresh Jagannathan. 2022. Specification-guided component-based synthesis from effectful libraries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 616–645.

[129] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. arXiv:2210.14306 [cs.SE]

[130] Bonnie A. Nardi and James R. Miller. 1990. The spreadsheet interface: A basis for end user programming. In *INTERACT*. North-Holland, 977–983.

[131] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3426428.3426919

[132] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

[133] OpenAI. 2023. ChatGPT. https://chat.openai.com/.

[134] OpenAI. 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.

[135] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

[136] Antti Oulasvirta and Pertti Saariluoma. 2006. Surviving task interruptions: Investigating the implications of long-term working memory theory. *International Journal of Human-Computer Studies* 64, 10 (2006), 941–961.

[137] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*. Springer, 1–19.

[138] Anjali Pal. 2021. Top 50 Bestselling Novels 2009-2020 of Amazon. https://www.kaggle.com/datasets/palanjali007/amazons-top-50-bestselling-novels-20092020

[139] John Paxton. 2002. Live Programming as a Lecture Technique. *J. Comput. Sci. Coll.* 18, 2 (dec 2002), 51–56.

[140] Hila Peleg. 2023. Program Synthesis Co-Design. (2023). https://doi.org/10.1184/R1/22277329.v1 Publisher: Plateau Workshop.

[141] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 159 (nov 2020), 30 pages. https://doi.org/10.1145/3428227

[142] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[143] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018-05-27) *(ICSE '18)*. Association for Computing Machinery, 1114–1124. https://doi.org/10.1145/3180155.3180189

[144] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? arXiv:2211.03622 [cs.CR]

[145] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310.

[146] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[147] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (jan 2019), 30 pages. https://doi.org/10.1145/3290385

[148] Aditya Ramachandran. 2022. Top IMDB Rated TV Shows. https://www.kaggle.com/datasets/adityaramachandran27/top-imdb-rated-tv-shows

[149] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. https://doi.org/10.22152/programming-journal.org/2019/3/9

[150] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 74–83.

[151] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[152] El Kindi Rezig, Anshul Bhandari, Anna Fariha, Benjamin Price, Allan Vanterpool, Vijay Gadepally, and Michael Stonebraker. 2021. DICE: Data Discovery by Example. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2819–2822. https://doi.org/10.14778/3476311.3476353

[153] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. https://doi.org/10.1145/3581641.3584037 arXiv:2302.07080 [cs].

[154] Mark Santolucito, William T Hallahan, and Ruzica Piskac. 2019. Live Programming By Example.

In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–4.

[155] Advait Sarkar, Andrew D. Gordon, Simon Peyton Jones, and Neil Toronto. 2018. Calculation view: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 85–93.

[156] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[157] Advait Sarkar, Sruti Srinivasa Ragavan, Jack Williams, and Andrew D. Gordon. 2022. End-user encounters with lambda abstraction in spreadsheets: Apollo's bow or Achilles' heel?. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11. https://doi.org/10.1109/VL/HCC53370.2022.9833131

[158] Thomas Schmitz, Dietmar Jannach, Birgit Hofer, Patrick Koch, Konstantin Schekotihin, and Franz Wotawa. 2017. A decomposition-based approach to spreadsheet testing and debugging. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 117–121. https://doi.org/10.1109/VLHCC.2017.8103458

[159] Christopher Schuster and Cormac Flanagan. 2016. Live programming by example: using direct manipulation for live program synthesis. In *LIVE Workshop*.

[160] Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *arXiv preprint arXiv:2003.09040* (2020).

[161] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290386

[162] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69.

[163] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 515–527.

[164] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 326–340.

[165] Sunbeom So and Hakjoo Oh. 2017. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*. Springer, 364–381.

[166] Gustavo Soares, Emerson Murphy-Hill, and Rohit Gheyi. 2013. Live feedback on behavioral changes. In *2013 1st International Workshop on Live Programming (LIVE)*. 23–26. https://doi.org/10.1109/LIVE.2013.6617344

[167] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[168] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (nov 2013), 64 pages. https://doi.org/10.1145/2490822

[169] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. https://doi.org/10.48550/arXiv.2302.02343 arXiv:2302.02343 [cs]

[170] Jamie Stark and Andrew Ireland. 1999. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, 44–51.

[171] TabNine. 2023. Code faster with AI completions. https://www.tabnine.com/.

[172] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S Păs ăreanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 899–913.

[173] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2023. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. Plateau Workshop.

[174] Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.

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

[176] Maryam Tohidi, William Buxton, Ronald Baecker, and Abigail Sellen. 2006. Getting the right design and the design right. In *CHI*. ACM, 1243–1252.

[177] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[178] Priyan Vaithilingam and Philip J Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs

from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 563–576.

[179] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2022-04-28) *(CHI EA '22)*. Association for Computing Machinery, 1–7. https://doi.org/10.1145/3491101.3519665

[180] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. 2022. Generation probabilities are not enough: Improving error highlighting for ai code suggestions. In *HCAI Workshop at NeurIPS*.

[181] Helena Vasconcelos, Matthew Jörke, Madeleine Grunde-McLaughlin, Tobias Gerstenberg, Michael Bernstein, and Ranjay Krishna. 2023. Explanations Can Reduce Overreliance on AI Systems During Decision-Making. http://arxiv.org/abs/2212.06823 arXiv:2212.06823 [cs].

[182] Sahil Verma and Subhajit Roy. 2017. Synergistic Debug-Repair of Heap Manipulations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 163–173.

[183] Bret Victor. 2012. Learnable Programming. (2012). http://worrydream.com/LearnableProgramming/

[184] Richard J. Waldinger and Richard C. T. Lee. 1969. PROW: A Step toward Automatic Program Writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 241–252.

[185] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

[186] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249

[187] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2023. Investigating and Designing for Trust in AI-powered Code Generation Tools. arXiv:2305.11248 [cs.HC]

[188] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[189] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–

357.

[190] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. In *26th International Conference on Intelligent User Interfaces*. Association for Computing Machinery, New York, NY, USA, 402–412. https://doi.org/10.1145/3397481.3450656

[191] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '97)*. ACM, New York, NY, USA, 258–265. https://doi.org/10.1145/258549.258721

[192] Mark Wilson-Thomas. 2023. Simplified Code Refinement and Debugging with GitHub Copilot Chat. https://devblogs.microsoft.com/visualstudio/simplified-code-refinement-and-debugging-with-github-copilot-chat/

[193] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

[194] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Vol. 00. 416–426. https://doi.org/10.1109/ICSE.2017.45

[195] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3187009.3177735

[196] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. EdSynth: Synthesizing API Sequences with Conditionals and Loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 161–171. https://doi.org/10.1109/ICST.2018.00025

[197] Dvora Yanow. 2017. Qualitative-interpretive methods in policy research. In *Handbook of public policy analysis*. Routledge, 431–442.

[198] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1657–1681.

[199] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 860–883.

[200] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer.

2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

[201] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 105, 16 pages. https://doi.org/10.1145/3411764.3445646

[202] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

[203] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces (IUI '22)*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120

[204] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. arXiv:2205.06537 [cs.SE]