LooPy: Interactive Program Synthesis with Control Structures

KASRA FERDOWSIFARD, UC San Diego, USA

SHRADDHA BARKE, UC San Diego, USA

HILA PELEG, Technion, Israel

SORIN LERNER, UC San Diego, USA

NADIA POLIKARPOVA, UC San Diego, USA

One vision for program synthesis, and specifically for programming by example (PBE), is an interactive programmer's assistant, integrated into the development environment. To make program synthesis practical for interactive use, prior work on Small-Step Live PBE has proposed to limit the scope of synthesis to small code snippets, and enable the users to provide local specifications for those snippets. This paradigm, however, does not work well in the presence of loops. We present LooPy, a synthesizer integrated into a live programming environment, which extends Small-Step Live PBE to work inside loops and scales it up to synthesize larger code snippets, while remaining fast enough for interactive use. To allow users to effectively provide examples at various loop iterations, even when the loop body is incomplete, LooPy makes use of *live execution*, a technique that leverages the programmer as an oracle to step over incomplete parts of the loop. To enable synthesis of loop bodies at interactive speeds, LooPy introduces *Intermediate State Graph*, a new data structure, which compactly represents a large space of code snippets composed of multiple assignment statements and conditionals. We evaluate LooPy empirically using benchmarks from competitive programming and previous synthesizers, and show that it can solve a wide variety of synthesis tasks at interactive speeds. We also performed a small qualitative user study which shows that LooPy's *block-level* specifications are easy for programmers to provide.

CCS Concepts: • **Human-centered computing** → **Graphical user interfaces**; • **Software and its engineering** → **Automatic programming**; **Programming by example**.

Additional Key Words and Phrases: Program synthesis, live programming

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) [Barke et al. 2020; Barman et al. 2015; Bavishi et al. 2019; Feng et al. 2018; Lubin et al. 2020; Shi et al. 2019; So and Oh 2017; Wang et al. 2017]; despite these advances, however, the programmer's assistant remains elusive. Where do existing PBE synthesizers fall short?

Authors' addresses: Kasra Ferdowsifard, UC San Diego, San Diego, CA, USA, kferdows@eng.ucsd.edu; Shraddha Barke, UC San Diego, San Diego, CA, USA, sbarke@eng.ucsd.edu; Hila Peleg, Technion, Haifa, Israel, hilap@cs.technion.ac.il; Sorin Lerner, UC San Diego, San Diego, CA, USA, lerner@cs.ucsd.edu; Nadia Polikarpova, UC San Diego, San Diego, CA, USA, npolikarpova@eng.ucsd.edu.

```python
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)

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

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

(b)

```python
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')
```

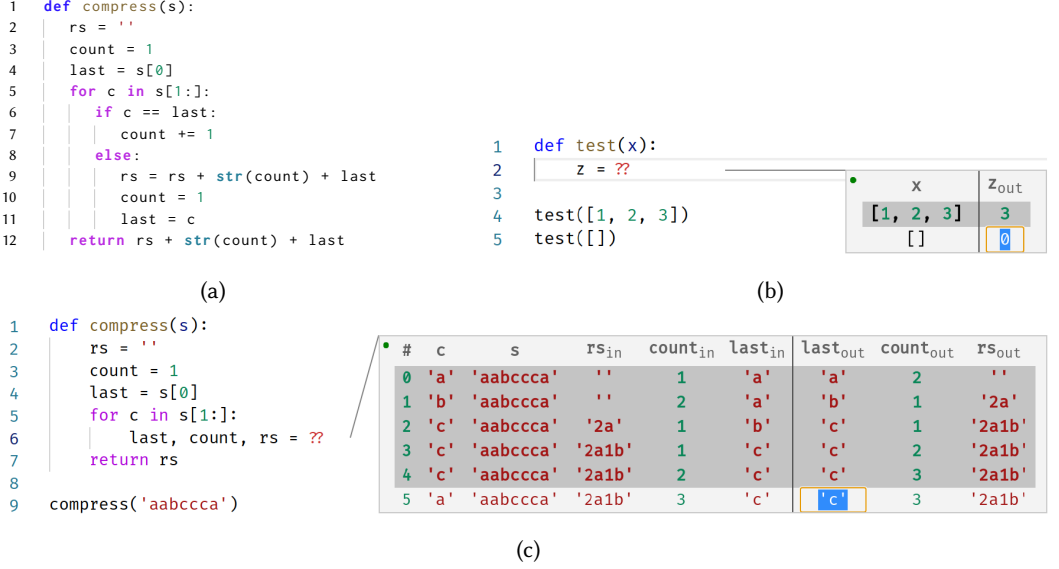| #	c	s	rs_{in}	$count_{in}$	$last_{in}$	$last_{out}$	$count_{out}$	rs_{out}
0	'a'	'aabccca'	''	1	'a'	'a'	2	''
1	'b'	'aabccca'	''	2	'a'	'b'	1	'2a'
2	'c'	'aabccca'	'2a'	1	'b'	'c'	1	'2a1b'
3	'c'	'aabccca'	'2a1b'	1	'c'	'c'	2	'2a1b'
4	'c'	'aabccca'	'2a1b'	2	'c'	'c'	3	'2a1b'
5	'a'	'aabccca'	'2a1b'	3	'c'	'c'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

(c)

Fig. 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.

Big-step synthesis. Consider a Python programmer who wants to solve the String Compression task from the popular book *Cracking the Coding Interview* [McDowell 2015, 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" → "2a1b3c1a", and expect the synthesizer to generate the body of the function compress in Fig. 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 [Shi et al. 2019] 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 [Ferdowsifard et al. 2020; Galenson et al. 2014] 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 expected to provide a local specification for that line. In particular, the interaction model we developed in [Ferdowsifard et al. 2020], dubbed *Small-Step Live PBE*, uses a

---

[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.

live programming environment—an enviroment where the program state is continuously displayed—named *Projection Boxes* [Lerner 2020]. 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. 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 [Peleg et al. 2020]: 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. 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. 1c. Given this input, LooPy generates the code on lines 6–11 of Fig. 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 Lubin et al. [2020]. 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. 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

---

[2]The code for LooPy is found at https://github.com/KasraF/LooPy
[3]Iteration counts are displayed in the "#" column of the projection box.

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.

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. 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).

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. 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 [Ferdowsifard et al. 2020], an approach that combines live programming in the Projection Boxes environment [Lerner 2020] 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. 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* $\sigma_{start} = \{x \mapsto [1, 2, 3], z \mapsto \bot\}$ into the *after-state* $\sigma_{end} = \{x \mapsto [1, 2, 3], z \mapsto 3\}$. If the user specifies z in both lines, the synthesis task becomes a set of examples (before-after pairs) $\sigma_{start}^0 \to \sigma_{end}^0, \sigma_{start}^1 \to \sigma_{end}^1$. 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. 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. 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.

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. 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 Lubin et al. [2020]), 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. 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]:

$$\sigma_{start}^0 = \{c \mapsto \text{'b'}, rs \mapsto \text{''}, count \mapsto 2, last \mapsto \text{'a'}\}$$

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):

$$\sigma_{end}^0 = \{c \mapsto \text{'b'}, \boxed{rs \mapsto \text{'2a'}}, count \mapsto 2, last \mapsto \text{'a'}\}$$

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):

$$\sigma_{start}^1 = \{\boxed{c \mapsto \text{'c'}}, rs \mapsto \text{'2a'}, \boxed{count \mapsto 1}, \boxed{last \mapsto \text{'b'}}\}$$

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

---

[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

Fig. 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).

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

Fig. 3. Specifying after-states for a hole *with multiple variables* in LOOPY.

Thanks to live execution, the two desirable properties of Live PBE are preserved inside the loop: (1) the before-states like $\sigma_{start}^0, \sigma_{start}^1$ 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 each evaluation of the hole *in order*; for example, in Fig. 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. 2c generates the following pairs:

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

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

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

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* [Albarghouthi et al. 2013; Udupa et al. 2013]. In this technique, an *expression enumerator* gradually builds more and more complex expressions by composing previously enumerated simpler expressions; all expressions are evaluated on the set of before-states σ_{start}^i, and expressions with the same output are pruned.
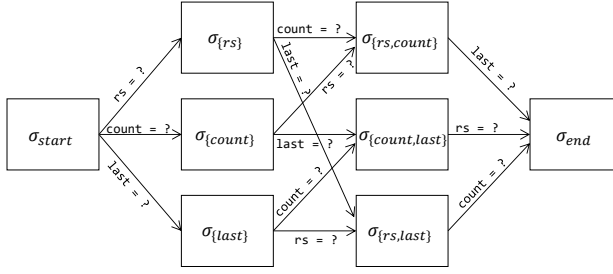
Fig. 4. Intermediate State Graph for synthesizing rs, last, and count.

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. 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, LooPy again takes less than a second to generate the sequence of assignments shown on the right.

```
9   rs += str(count) + last
10  count = 1
11  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.

Specifically, consider the before-after pair for iteration 1 in Fig. 3:

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

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

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 $\sigma_{\{rs\}}$); similarly, the second state is the one where only rs and count are updated (we denote it $\sigma_{\{rs,count\}}$). Hence our synthesis problem has been reduced to three independent sub-problems, $\sigma_{start} \rightarrow \sigma_{\{rs\}}$, $\sigma_{\{rs\}} \rightarrow \sigma_{\{rs,count\}}$, and $\sigma_{\{rs,count\}} \rightarrow \sigma_{end}$, 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 $\sigma_{\{last\}}$, and likewise both rs, last, count and last, rs, count pass through $\sigma_{\{rs,last\}}$.

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. 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 $\sigma_{\{rs\}}$ encounters the expression c, this expression gets evaluated in the state $\sigma_{\{rs\}}$ and tested as a possible assignment for all outgoing edges from that node. In our example, the assignment last = c transforms $\sigma_{\{rs\}}$ into $\sigma_{\{rs,last\}}$, hence we label the outgoing edge last from $\sigma_{\{rs\}}$ 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 $\sigma_{start}, \sigma_{\{rs\}}, \sigma_{\{rs,count\}}, \sigma_{end}$.

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. 1c. Again, in less than a second, LooPy replaces the hole with the code shown on the right. To understand how LooPy generates this code

```
6  if c == last:
7      count += 1
8  else:
9      rs += str(count) + last
10     count = 1
11     last = c
```

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 [Alur et al. 2017], 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 iterations in Fig. 1c, where $\epsilon^i = \sigma^i_{start} \to \sigma^i_{end}$ for $i = 0, 1$ and

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

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

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. 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 $\sigma_{\{count\}}$. 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: $\langle\{\epsilon^0, \epsilon^1\}, \emptyset\rangle$ and $\langle\{\epsilon^0\}, \{\epsilon^1\}\rangle$. 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 $\langle\{\epsilon^0\}, \{\epsilon^1\}\rangle$, 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

---

[6]For simplicity, here we ignore Python's simultaneous assignment statements, which update multiple variables at a time. In Sec. 4 and Sec. 5.3 we show how LooPy synthesizes such assignments.
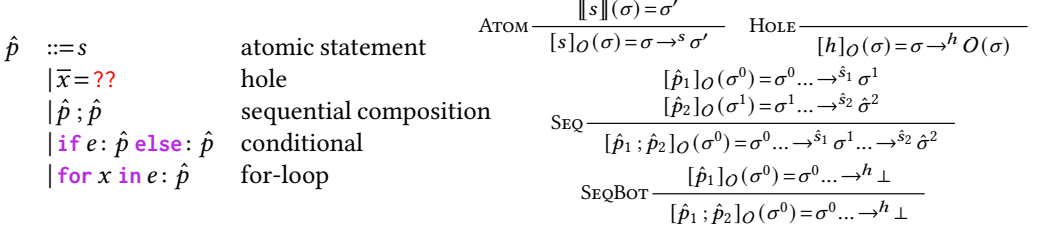
$$\hat{p} \quad ::= s \qquad\qquad \text{atomic statement}$$

$$\text{ATOM} \frac{[\![s]\!](\sigma)=\sigma'}{[s]_O(\sigma)=\sigma\rightarrow^s\sigma'} \qquad \text{HOLE} \frac{}{[h]_O(\sigma)=\sigma\rightarrow^h O(\sigma)}$$

$$| \overline{x} = \text{??} \qquad\qquad \text{hole}$$

$$| \hat{p}\,;\hat{p} \qquad\qquad \text{sequential composition}$$

$$| \text{if } e\colon \hat{p} \text{ else}\colon \hat{p} \quad \text{conditional}$$

$$| \text{for } x \text{ in } e\colon \hat{p} \quad \text{for-loop}$$

$$\text{SEQ} \frac{[\hat{p}_1]_O(\sigma^0)=\sigma^0...\rightarrow^{\hat{s}_1}\sigma^1 \qquad [\hat{p}_2]_O(\sigma^1)=\sigma^1...\rightarrow^{\hat{s}_2}\hat{\sigma}^2}{[\hat{p}_1\,;\hat{p}_2]_O(\sigma^0)=\sigma^0...\rightarrow^{\hat{s}_1}\sigma^1...\rightarrow^{\hat{s}_2}\hat{\sigma}^2}$$

$$\text{SEQBOT} \frac{[\hat{p}_1]_O(\sigma^0)=\sigma^0...\rightarrow^h \bot}{[\hat{p}_1\,;\hat{p}_2]_O(\sigma^0)=\sigma^0...\rightarrow^h \bot}$$

Fig. 5. (Left) The syntax of sketches $\hat{p}$ in LooPy; programs p use the same grammar excluding holes. (Right) Definition of a live trace $[\hat{p}]_O(\sigma)$ of a sketch $\hat{p}$ starting from store σ using a live execution oracle O; rules for conditionals and loops are omitted for brevity.

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.

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. 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 $[\![e]\!]\colon \text{Store}\rightarrow\text{Val}$ and $[\![s]\!]\colon \text{Store}\rightarrow\text{Store}$ (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 $\overline{x} = \text{??}$, 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 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 O only updates the output variables of the hole, *i.e.* for a sketch with the hole $\overline{x}=\text{??}$ and for any store σ and program variable $y\notin\overline{x}$, $O(\sigma)(y)=\sigma(y)$.

Using the oracle, we define a *live trace* $[\hat{p}]_O(\sigma^0)$ 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. 5 (right). 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 $\sigma\rightarrow^{s_1}...\rightarrow^{s_n}\sigma''$, 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 $\langle\hat{p},O,\sigma^0\rangle$ of a sketch $\hat{p}$, a live execution oracle 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) $[p](\sigma^0) \prec [\hat{p}]_O(\sigma^0)$, 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), and also serves as the live execution oracle O. The LooPy UI incrementally builds the live trace $[\hat{p}]_O(\sigma^0)$ by querying the oracle via projection boxes, as shown in Fig. 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", $O(\sigma^1)$ is taken to be $\bot$, 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 $\langle \hat{p}, O, \sigma^0 \rangle$, 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 $\mathcal{E}$, 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 $\sigma_{start} \rightarrow \sigma_{end} \in \mathcal{E}$, $[p^*](\sigma_{start}) = \sigma_{start} \cdots \rightarrow^s \sigma_{end}$.

To reduce the live PBE task $\langle \hat{p}, O, \sigma^0 \rangle$ to a block-level task, we define $\mathcal{E} = \{\sigma \rightarrow \sigma' \mid \sigma \rightarrow^h \sigma' \in [\hat{p}]_O(\sigma^0)\}$. 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 $\mathcal{E}$, then $\hat{p}[p^*/h]$ is a solution to original live PBE task $\langle \hat{p}, O, \sigma^0 \rangle$.

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. 4 focuses on straight-line code blocks (sequences of assignments); then Sec. 5 extends the synthesis algorithm to conditionals.

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:

$$p ::= \overline{x} = \overline{e} \mid p ; p$$

Here $\overline{x} = \overline{e}$ 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 $\{x \mapsto 1, y \mapsto 1\}$ yields the store $\{x \mapsto 2, y \mapsto 3\}$.

We begin our exposition in Sec. 4.1 with the simplest version of the algorithm, where the synthesis task is restricted to a single example ($|\mathcal{E}| = 1$), and the solution is restricted to a single (simultaneous) assignment. Sec. 4.2 then extends the algorithm to generate a sequence of assignments, and Sec. 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* [Albarghouthi et al. 2013; Udupa et al. 2013]. 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. 4.1–4.2, we can think of an expression enumerator as a black-box function $\mathrm{Enum}(\sigma)$, 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, $\mathrm{Vars}(\sigma)$, 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 $\mathrm{Enum}(\sigma)$ 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.

4.1 Single Example, Single Assignment

We begin by examining the case where $\mathcal{E} = \{\epsilon\}$ and the target program contains a single assignment. We will denote Mod_ϵ the set of variables *modified* by an example $\epsilon = \sigma_{start} \to \sigma_{end}$, *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.

Synthesis algorithm. The synthesis algorithm maintains a *variable assignment* $\mathcal{A}$, which maps each variable in Mod_ϵ to an expression or $\bot$; the assignment is initialized by setting $\mathcal{A}(x) = \bot$ for every $x \in \mathrm{Mod}_\epsilon$. The algorithm then creates a single expression enumerator $\mathrm{Enum}(\sigma_{start})$. In each iteration, it draws another expression e_i from the enumerator stream, and for every unassigned variable x (such that $\mathcal{A}(x) = \bot$), it tests whether e_i is *valid* for x, *i.e.* whether $[\![e_i]\!](\sigma_{start}) = \sigma_{end}(x)$; if so, the algorithm updates $\mathcal{A}(x)$ to e_i. When $\mathcal{A}$ is *complete* (*i.e.* $\mathcal{A}(x) \neq \bot$ for every x), the algorithm terminates and returns $\overline{x} = \overline{\mathcal{A}(x)}$ 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 $\overline{x} = \overline{e}$ are evaluated in the same store σ_{start}.

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

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

---

[7]In Sec. 4.3 we will generalize the notion of expression enumerators from a single store to a vector of stores.

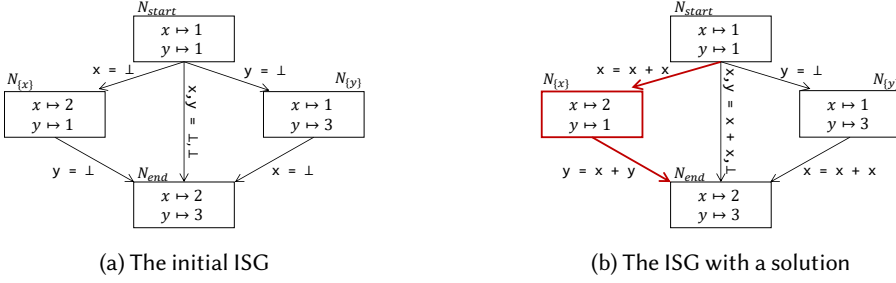(a) The initial ISG   (b) The ISG with a solution

Fig. 6. The Intermediate State Graph for Ex. 1. The path of complete edges is marked in red.

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: $\sigma_{start} \to \sigma_{\{x\}}$, which transforms the start state into an *intermediate state* where only x has been updated, and $\sigma_{\{x\}} \to \sigma_{end}$, which transforms the intermediate state into the end state. Each sub-task can then be solved independently using the algorithm from Sec. 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 $X \subset \mathsf{Mod}_\epsilon$, let a *partially-updated state* σ_X be the state where only those variables in X have been updated:

$$\sigma_X \;=\; \{x \mapsto \sigma_{end}(x) \,|\, x \in X\} \,\cup\, \{x \mapsto \sigma_{start}(x) \,|\, x \in \mathsf{Mod}_\epsilon \setminus X\}$$

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

Example 2. In Ex. 1, there are two possible intermediate states to consider: $\sigma_{\{x\}} = \{x \mapsto 2, y \mapsto 1\}$ and $\sigma_{\{y\}} = \{x \mapsto 1, y \mapsto 3\}$. A solution p^* can transition from σ_{start} to σ_{end} through $\sigma_{\{x\}}$, where x is modified first, through $\sigma_{\{y\}}$, 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 $\{\epsilon\} = \{\sigma_{start} \to \sigma_{end}\}$, its ISG is a directed acyclic graph where:
   (1) there is a node N_X for each $X \subset \mathsf{Mod}\epsilon$, 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 $\mathcal{A}_{(X,X')}$, whose domain is $X' \setminus X$.

Example 3. Fig. 6 depicts the ISG for Ex. 1 with different variable assignments $\mathcal{A}_E$ on the edges: on the left all the assignments are empty; on the right, some (but not all) the assignments are complete.

Synthesis algroithm. The synthesis algorithm maintains an ISG, where the assignment for each edge E is initialized with $\mathcal{A}_E(x) = \bot$ 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 $\mathcal{A}_{(X,X')}(x) = \bot$), 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 $\mathcal{A}_{(X,X')}(x)$ to e_i. When all variables on an edge are assigned (*i.e.* $\mathcal{A}_{(X,X')}(x) \neq \bot$ 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. 6a. 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 $[\![x + x]\!](\sigma_{start}) = 2$, is then tested for validity for every variable on each of the three outgoing edges from N_{start}:

– $(N_{start}, N_{\{x\}})$, variable x: $\sigma_{\{x\}}(x) = 2$, so $\mathcal{A}_{(start,\{x\})}(x)$ is set to x + x.
– $(N_{start}, N_{\{y\}})$, variable y, $\sigma_{\{y\}}(y) \neq 2$, so $\mathcal{A}_{(start,\{x\})}(y)$ remains $\bot$.
– (N_{start}, N_{end}), variables x and y: $\sigma_{end}(x) = 2$ but $\sigma_{end}(y) \neq 2$, so $\mathcal{A}_{(start,end)}(x)$ is set to x + x, and $\mathcal{A}_{(start,end)}(y)$ 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 $[\![x + y]\!](\sigma_{\{x\}}) = 3$ is tested for validity for the variable y of the sole outgoing edge $(N_{\{x\}}, N_{end})$ from $N_{\{x\}}$. Since $\sigma_{end}(y) = 3$, $\mathcal{A}_{(\{x\},end)}(y)$ is set to x + y, and this edge is now complete. Moreover, as shown in Fig. 6b, 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. 6 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 $\mathsf{Enum}(\sigma_{\{x\}})$ yields both x and y (and larger expressions build from both of these variables), since the two variables are not equivalent in $\sigma_{\{x\}}$.

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 $\sum_{x \in X' \setminus X} \text{size}(\mathcal{A}_{(X,X')}(x))$, where size is the size of an expression in AST nodes (with $\text{size}(\bot) = \infty$). 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. 6 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.

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 our synthesis algorithm to this setting. More specifically, let $\mathcal{E} = \langle \epsilon^1, ..., \epsilon^n \rangle$ 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 $\text{Mod}_{\mathcal{E}}$ to include variables modified in any of the examples: $\text{Mod}_{\mathcal{E}} = \bigcup_{\epsilon \in \mathcal{E}} \text{Mod}_\epsilon$.

Expression enumerators. An expression enumerator $\text{Enum}(\langle \sigma^1, ..., \sigma^n \rangle)$ 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* $\langle v^1, ..., v^n \rangle$; 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:

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

Importantly, the topology of the graph is unchanged, in the sense that the set of nodes and edges is determined only by $\text{Mod}_{\mathcal{E}}$ 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 $\text{Enum}(\langle \sigma_X^1, ... \sigma_X^n \rangle)$. 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.

5 SYNTHESIZING CONDITIONALS

In this section, we extend our block-level synthesis algorithm to support conditional statements. We begin in Sec. 5.1 with a restricted setting where the solution has a single assignment in each branch of the conditional; Sec. 5.2 extends the algorithm to combine conditionals with assignment sequences.

5.1 Single Conditional Assignment

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

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

---

[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.

Unlike an unconditional assignment $x = e$, where we had to find a single expression e that is valid *for all* examples $\mathcal{E}$, 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 $\mathcal{E}_{then} \subset \mathcal{E}$;
(2) e_{else} is valid for the rest of the example $\mathcal{E}_{else} = \mathcal{E} \setminus \mathcal{E}_{then}$; and
(3) e_{cond} is a boolean expression that separates these two subsets, *i.e.* evaluates to `True` on all $\mathcal{E}_{then}$ and to `False` on all $\mathcal{E}_{else}$.

The general idea behind the synthesis algorithm is to use a single enumerator $\mathsf{Enum}(\langle \sigma^1_{start}, ..., \sigma^n_{start} \rangle)$ 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 $\mathcal{E}$ into $\mathcal{E}_{then}$ and $\mathcal{E}_{else}$, the algorithm must consider all possible partitions $\pi = \langle \mathcal{E}_1, \mathcal{E}_2 \rangle$. Note, however, that a solution for $\langle \mathcal{E}_1, \mathcal{E}_2 \rangle$, can be transformed into a solution for the symmetric partition $\langle \mathcal{E}_2, \mathcal{E}_1 \rangle$, 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 $\Pi_{\mathcal{E}}$, with $|\Pi_{\mathcal{E}}| = 2^{|\mathcal{E}|-1}$.

Example 5. Consider a synthesis task $\mathcal{E} = \{\epsilon^1, \epsilon^2\}$ where $\epsilon^1 = \{x \mapsto -1\} \rightarrow \{x \mapsto 1\}$ and $\epsilon^2 = \{x \mapsto 2\} \rightarrow \{x \mapsto 2\}$. There are four partitions of $\mathcal{E}$:

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

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 $\Pi_{\mathcal{E}} = \{\pi_1, \pi_2\}$.

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

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

Example 6. Fig. 7 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 $\pi \in \Pi_{\mathcal{E}}$. 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 $\mathcal{A}^{\mathcal{E}_j}(x)$ and $C(\pi)$ to $\bot$, except $C(\langle \mathcal{E}, \emptyset \rangle) \mapsto \mathsf{True}$. In each iteration, the algorithm draws one expression e_i from the enumerator and updates the state as follows:

(1) For each partition $\langle \mathcal{E}_1, \mathcal{E}_2 \rangle \in \Pi_{\mathcal{E}}$, the algorithm tests whether e_i is valid over either of the $\mathcal{E}_j$; in that case, $\mathcal{A}^{\mathcal{E}_j}(x)$ is updated to e^i unless already set to something other than $\bot$.
(2) If e_i is a boolean expression that evaluates without errors on all examples, the algorithm searches for a partition $\pi \in \Pi_{\mathcal{E}}$ such that e_i matches π; if found, $C(\pi)$ is updated to e_i, unless already set. Note that a matching partition π might not exist in $\Pi_{\mathcal{E}}$, since $\Pi_{\mathcal{E}}$ only stores half
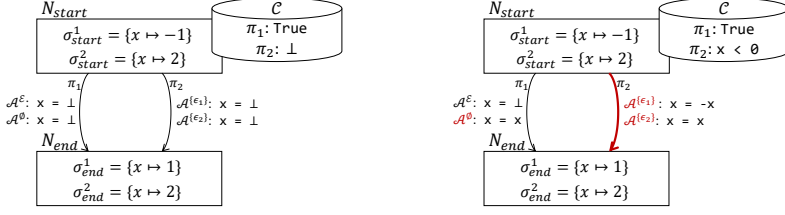
Fig. 7. Initial (left) and final (right) state of the synthesis algorithm for the task in Ex. 5. Complete variable assignments and complete edges (partitions) are highlighted in red. On the right, partition π_2 is complete, because both of its variable assignments are complete, and it has a condition in C.

of the partitions; in this case, there must be a $\pi' \in \Pi_{\mathcal{E}}$ 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 C accordingly.

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

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

Example 7. For the synthesis task in Ex. 5, the state is initialized as shown in Fig. 7 (left). The first iteration enumerates expression x, which is valid for the example ϵ^2, and hence we update $\mathcal{A}^{\{\epsilon^2\}}(x) = x$ (and also, trivially, $\mathcal{A}^{\emptyset}(x) = x$). A later iteration enumerates -x, which is valid for ϵ^1, updating $\mathcal{A}^{\{\epsilon^1\}}(x) = $ -x. At this point, the partition $\pi_2 = \langle \{\epsilon^1\}, \{\epsilon^2\} \rangle$ has both of its variable assignments complete, but the partition itself is not yet complete, since it does not have a condition in 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 $C(\pi_2) = $ x < 0. At this point, π_2 is complete, and the algorithm terminates. This final state is depicted in Fig. 7 (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. 4.2, we pick the solution with the smallest overall size in AST nodes, *i.e.* minimizing size(e_{cond})+size(e_{then})+ size(e_{else}). For the partition $\langle \mathcal{E}, \emptyset \rangle$, the size is computed simply as size(e_{then}), since this solution can be simplified into an unconditional assignment.

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. 4.2 to handle sequential composition with partitions from Sec. 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:

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

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:

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

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. 7. In general, to turn an ISG into a conditional ISG we simply clone every edge $2^{|\mathcal{E}|-1}$ times (one for each partition), and associate two variable assignments (instead of one) with each edge. We also add a single condition store C, associated with the node N_{start}.

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

(1) there is a node N_X for each $X \subset \text{Mod}_{\mathcal{E}}$, which represents the vector of partially-updated stores $\langle \sigma_X^1, ..., \sigma_X^n \rangle$;

(2) for each pair of states $N_X, N_{X'}$ such that $X \subsetneq X'$, and for each partition $\pi \in \Pi_{\mathcal{E}}$, there is an edge $(N_X, N_{X'})_\pi$;

(3) each edge $(N_X, N_{X'})_\pi$ where $\pi = \langle \mathcal{E}_1, \mathcal{E}_2 \rangle$ is labeled with a pair of variable assignments $\mathcal{A}_{(X,X')}^{\mathcal{E}_1}$ and $\mathcal{A}_{(X,X')}^{\mathcal{E}_2}$, whose domain is $X' \setminus X$.

(4) the node N_{start} is labeled with a condition store C.

Given a conditional ISG $\mathcal{G}$ and a partition π, we denote $\mathcal{G}/\pi$ the sub-graph of $\mathcal{G}$ spanned by all the edges of the form $(N, N')_\pi$. Each $\mathcal{G}/\pi$ 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. 4.2 and Sec. 5.1: namely, e_i is tested for validity for both $\mathcal{E}_1$ and $\mathcal{E}_2$, for each outgoing edge $(N_X, N_{X'})_{\langle \mathcal{E}_1, \mathcal{E}_2 \rangle}$, and each variable $x \in X' \setminus X$, and the assignment $\mathcal{A}_{(X,X')}^{\mathcal{E}_j}$ 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 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 C, and the conditions are produced by the enumerator at N_{start}.

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

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

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

5.3 Post-Processing

Because the synthesis algorithm described in Sec. 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 fixed point:
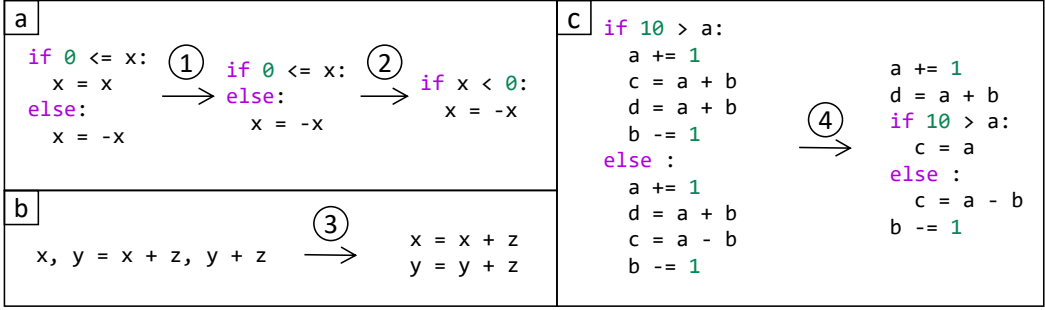
Fig. 8. Examples of post-processing of synthesized conditional programs to make them more readable.

(1) **Removing self-assignment.** A variable $x \in \mathrm{Mod}_\mathcal{E}$ 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. 8 (a).

(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. 8 (a).

(3) **Splitting simultaneous assignments.** The generated solutions might include simultaneous assignments even when they are not strictly required (recall the example in Sec. 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. 8 (b) shows an example of this rewrite; this is sound because x + z does not mention y and y + z does not mention x. On the other hand, x, y = x + z, y + x cannot be straightforwardly split, because the right-hand side assigned to y mentions x. 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. 8 (c). 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. 8 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.

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?

Benchmark suite	Number of benchmarks	Average $|\text{Mod}_{\mathcal{E}}|$	Avg solution size (AST nodes)	Average 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

Table 1. Summary of the four benchmark suites used in LooPy's empirical evaluation.

(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. 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 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. 7.
(4) *FrAngel:* 23 benchmarks from FrAngel's *ControlStructures* benchmark suite [Shi et al. 2019].

The statistics for these benchmarks, including the size of specification provided, are shown in Tab. 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.

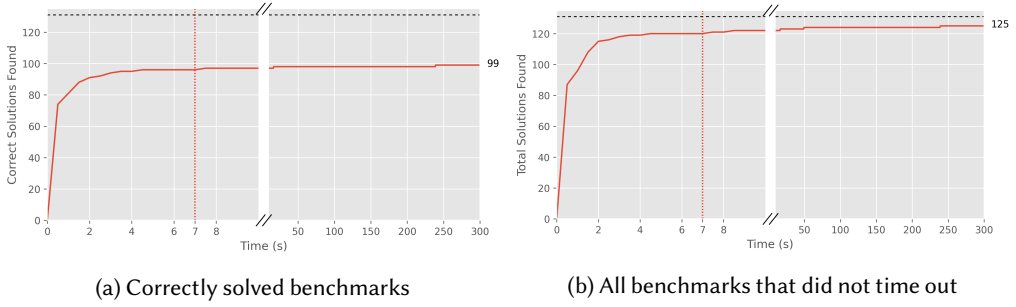(a) Correctly solved benchmarks

(b) All benchmarks that did not time out

Fig. 9. Performance of the LooPy synthesizer on the full benchmark suite at 300s timeout.

(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.

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

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. 9.

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 context of their work. We therefore **answer RQ1 in the affirmative**.

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 *ControlStructures* 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.

6.2.1 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
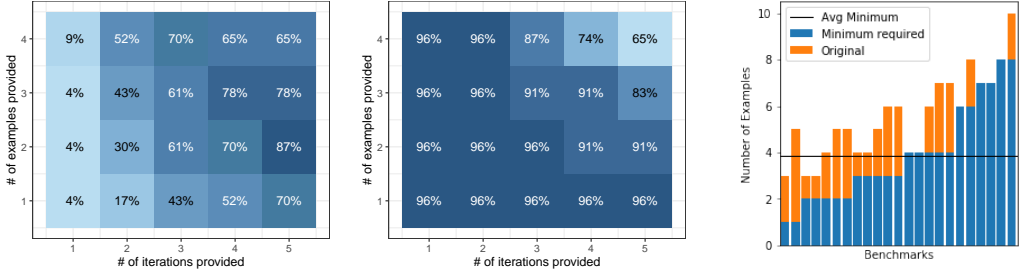
(a) Correctly solved benchmarks (b) All benchmarks that did not time out (c) FʀAɴɢᴇʟ end-to-end examples

Fig. 10. The effect of the number of examples and the number of iterations from each example on LooPʏ using the FʀAɴɢᴇʟ benchmark set, and the number of examples provided and required by FʀAɴɢᴇʟ.

FʀAɴɢᴇʟ 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. 10c. For most benchmarks, FʀAɴɢᴇʟ 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.

6.2.2 Varying number of examples for LooPʏ.
To measure the size of user input LooPʏ needs, we exercised it with varying numbers of examples and loop iterations per example.

We translated each FʀAɴɢᴇʟ 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 FʀAɴɢᴇʟ 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 FʀAɴɢᴇʟ benchmark file.

For each FʀAɴɢᴇʟ benchmark, we run LooPʏ 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 LooPʏ found the correct program appear in Fig. 10.

Results. LooPʏ'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 FʀAɴɢᴇʟ 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 LooPʏ, 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, LooPʏ succeeds on all FʀAɴɢᴇʟ benchmarks but one.

It is also rather expected that providing just one iteration has a low success rate, even with multiple examples (see the leftmost column of Fig. 10a). It is similarly unsurprising that LooPʏ times out more often as the number of examples increases (see Fig. 10b), due to the properties of Observational Equivalence we discussed in Sec. 4.2.

(a) Time: single assignment   (b) Time: assignment sequence   (c) Correctness
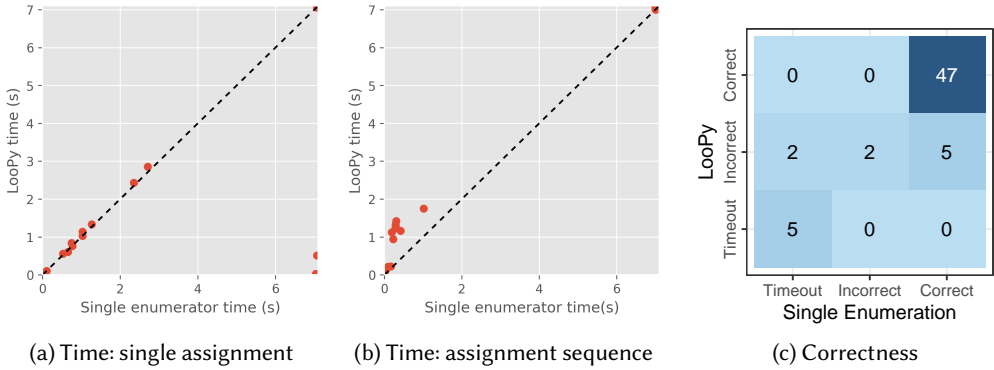
Fig. 11. Differences in synthesis time and correctness between LooPy and a single expression enumerator.

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. 7. With this in mind, we **answer RQ2 in the affirmative.**

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. 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. 11.

Results. We first focus on synthesis times for those benchmarks that only require a single assignment (Fig. 11a). 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. 11b), 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.
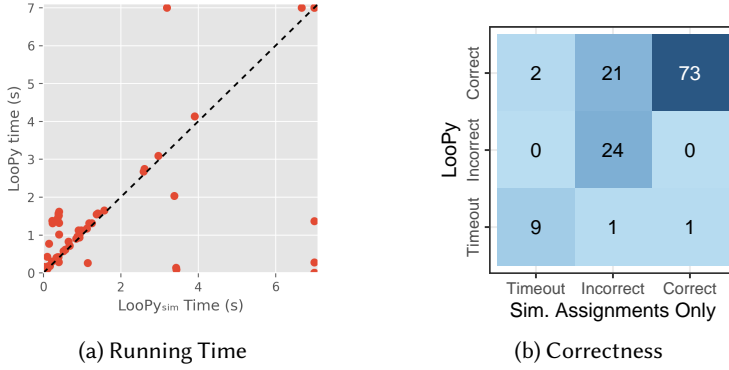
(a) Running Time   (b) Correctness

Fig. 12.   Differences in running time and correctness between LooPy and LooPy$_{sim}$.

There are five benchmarks that were correctly solved by the baseline synthesizer and incorrectly solved by LooPy (Fig. 11c); all five require multiple assignments. In three of those, LooPy finds a different order of assignments, which happen 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 benchmarks without control flow, but **this effect is small compared to the wealth of new benchmarks that can now be solved.**

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. 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$_{sim}$, a version of LooPy capable of synthesizing only simultaneous assignments but not assignment sequences. LooPy$_{sim}$ maintains a conditional ISG with only two nodes, N_{start} and N_{end}. We ran LooPy$_{sim}$ on our entire benchmark suite and measured time to termination and correctness, compared to the original LooPy synthesizer. The results appear in Fig. 12.

Results. When it comes to synthesis times, on average, LooPy tends to be slightly slower that LooPy$_{sim}$, 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$_{sim}$ finishes and LooPy times out. There are also two benchmarks where the opposite happens: LooPy finishes and LooPy$_{sim}$ 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. 12b shows that there are 21 benchmarks that LooPy solves correctly and LooPy$_{sim}$ 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$_{sim}$ 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.

---

[9]In Fig. 12a there appear to be three such benchmarks, but one of them actually finishes right before the timeout.

```
1   """
2   k-th digital root:                           1   """
3       Compute the k-th digital root of a natural    2   Extract Numbers:
4       number n, that is, replace n with the sum     3       Given a string containing numbers separated by
5           of its digits k times.                     4       exactly one non-integer character, return a
6       >>> task(1798, 3)                          5       list containing all the numbers in the string.
7       7                                          6       >>> task('13a7b42')
8       (because                                   7       [13, 7, 42]
9           1. 1 + 7 + 9 + 8 -> 25                 8   """
10          2. 2 + 5         -> 7                  9   def task(s):
11          3. 7             -> 7)                 10      rs = []
12  """                                           11      return rs
13  def task(n, k):                               12
14      rs = n                                    13  task('13a7b42')
15      return rs
16
17  task(1798, 3)
```

Fig. 13. The initial state of the study tasks, as provided to the users in VSCode.

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**.

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 question:

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

Implementation. We modified our VSCode extension for Small-Step Live PBE [Ferdowsifard et al. 2020], 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.

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. 6. In the first part of the study, users watched a tutorial video and solved a training task (String Compression from Sec. 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. 13) 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 [Ferdowsifard et al. 2020].

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, therefore invalidating 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. 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 [Ferdowsifard et al. 2020]. 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.

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 [Shi et al. 2019], 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 [Albarghouthi et al. 2013], Myth [Osera and Zdancewic 2015], Smyth [Lubin et al. 2020], λ^2 [Feser et al. 2015], Bigλ [Smith and Albarghouthi 2016], and RESL [Peleg et al. 2020]. 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 [Chasins et al. 2018], 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 [Alur et al. 2015; Kneuss et al. 2013; Leino and Milicevic 2012; Polikarpova et al. 2016; Shi et al. 2019]. It is most related to the the approach used by EUSolver [Alur et al. 2017], 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 [Gulwani et al. 2011] 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 [Ferdowsifard et al. 2020] and CodeHint [Galenson et al. 2014]; various other Python synthesizers, for example TFCoder [Shi et al. 2020], AutoPandas [Bavishi et al. 2019], Wrex [Drosos et al. 2020]; and synthesizers for other languages [Feng et al. 2017; Galenson et al. 2014; Gvero et al. 2013; James et al. 2020; Mandelin et al. 2005; Yang et al. 2018]. 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 [Udupa et al. 2013] and Escher [Albarghouthi et al. 2013], and is used in many synthesizers [Barke et al. 2020; Peleg et al. 2020; Peleg and Polikarpova 2020; Shi et al. 2020]. 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 Omar et al. [2019] and further adapted by Lubin et al. [2020], 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.

ACKNOWLEDGMENTS

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification*. Springer, 934–950.

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.

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.

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 (11 2020).

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.

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

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.

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

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

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL*.

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 event-place: Virtual Event, USA.

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.

Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *PLDI*. ACM, 62–73.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI*.

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. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 205:1–205:27.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *OOPSLA*. ACM, 407–426.

K. Rustan M. Leino and Aleksandar Milicevic. 2012. Program Extrapolation with Jennisys. In *OOPSLA*. ACM, 411–430.

Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming *(CHI '20)*. ACM. https://doi.org/10.1145/3313831.3376494

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.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*.

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

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

Hila Peleg, Roi Gabai, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (11 2020).

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*.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI*. ACM, 522–538.

Kensen Shi, David Bieber, and Rishabh Singh. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. *arXiv preprint arXiv:2003.09040* (2020).

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

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340.

Sunbeom So and Hakjoo Oh. 2017. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*. Springer, 364–381.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

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.

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)*. IEEE, 161–171.