Focused Live Programming with Loop Seeds

Sorin Lerner
University of California, San Diego
lerner@cs.ucsd.edu

ABSTRACT

Live programming is a paradigm in which the programmer can visualize the runtime values of the program each time the program changes. The promise of live programming depends on using test cases to run the program and thereby provide these runtime values. In this paper we show that in some situations test cases are insufficient in a fundamental way, in that there are no test inputs that can drive certain incomplete loops to produce useful data, a problem we call the loop-datavoid problem. The problem stems from the fact that useful data inside the loop might only be produced after the loop has been fully written. To solve this problem, we propose a paradigm called Focused Live Programming with Loop Seeds, in which the programmer provides hypothetical values to start a loop iteration, and then the programming environment focuses the live visualization on this hypothetical loop iteration. We introduce the loop-datavoid problem, present our proposed solution, explain it in detail, and then present the results of a user study.

Author Keywords

Live Programming; Program Testing; Debugging.

CCS Concepts

•Human-centered computing → Human computer interaction (HCI);

INTRODUCTION

Live programming [7, 20, 21, 10, 1, 12, 11, 17] is a paradigm in which the programmer can visualize the runtime values of the program each time the program changes. Live programming has been shown to help programmers find mistakes quickly, as they are writing the code [22, 13].

The biggest promise of live programming lies in its ability to provide feedback *while* writing code, not just after the code is written. To make this promise a reality, live programming requires that data be shown to the programmer *while* the code is being written. The most common approach for producing this data involves executing the (incomplete) program on some *input values*. These input values are typically provided by the programmer as inputs to the entire program or to the function/method being written.

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10.
http://dx.doi.org/10.1145/3379337.3415834

In this paper we show that such input values are insufficient for imperative programs, in a fundamental way. Indeed there are cases where input values will *never* drive an incomplete imperative program to show useful live data in a loop. This is because the data that is useful for writing the code inside a loop is generated by the loop itself, which does not happen until the loop has been fully written. In essence, until the loop is written in full, the live data in the loop is either unavailable, incomplete or incorrect, which not only defeats the purpose of live programming, but worse yet actually leads to programmers being confused. This situation can arise in a such fundamental settings as: insertion sort, Dijkstra's algorithm for shortest path, building histograms, reversing a list, and building an interpreter.

As such, this paper shows that to see useful live data while writing loops, in addition to *input values*, programmers must sometimes provide what we call *loop seeds*. Loop seeds are values that are used to start a *hypothetical* loop iteration. These are *hypothetical* in the sense that the loop iteration is not something that can currently be reached in the program, because the program is incomplete. The programmer can then focus on this hypothetical loop iteration to write the loop body, thus only seeing the live data for this one hypothetical loop iteration. When the loop body is finished the programmer can unfocus from the hypothetical loop iteration to fall back into seeing the program's behavior for all loop iterations on the test inputs. As such, we call this approach *Focused Live Programming with Seeds* (FLiPS).

In summary, our contributions are as follows:

- We show that in some cases test inputs at function/method/program boundaries are not sufficient to produce useful live data for writing loops. We call this the *loop-datavoid* problem.
- To solve this problem, we introduce the notion of *loop seeds*, which are values used to start a hypothetical loop iteration, thus producing data for the programmer to visualize. One important point of this paper is not that *loop seeds* are just helpful, but that they are in some cases a necessity without which live programming can loose many of its benefits.
- We present an approach called *Focused Live Programming with Seeds* (FLiPS) that uses loops seeds to focus the programmer's attention on a single hypothetical loop iteration.
- We evaluate FLiPS through a user study, and show that FLiPS is helpful, easy-to-use, and that it significantly increases the availability of live data while writing loops. Finally, we also observe that FLiPS actually provides benefits beyond the original ones we anticipated.

RELATED WORK

Live programming is an area of research whose history dates back to the seminal work of Hancock [7] that introduced many important concepts in live programming. Since then, a variety of live environments has been developed for various languages, including Python [6, 10], Java [3], Javascript [17, 1], Lisp [2] and ML-like languages [16]. As the field developed, several essays were published on categorizing different kinds of liveness [19, 21], and several studies explored the benefits of live programming [22, 13]. Most recently, there has also been work on live editing the output of a program through direct manipulation [8, 15, 9].

Broadly speaking, our work is different from prior live programming research in three ways: (1) we identify the *loop-datavoid* problem, which states that, for certain loops, test inputs are not enough to provide useful live data while the loop is being written (2) we provide a solution to this problem in the form of loop seeds so the programmer can see live data for a hypothetical loop iteration (3) we reify loop seeds in a new programming paradigm called Focused Live Programming with Seeds (FLiPS).

Our work is most related to the idea from live programming of focusing on a single loop iteration. For example, the Babylonian live editor [17, 18] supports sliders on loops to focus on a particular loop iteration. However, this prior work does not identify the loop-datavoid problem and does not have the concept of loop seeds. Instead, programmers can only provide test inputs at method boundaries, which means they can only focus on loop iterations that are currently reached, and as we will show this does not work well for incomplete loops.

The Omnicode [10] system, which makes the entire history of program execution available to the programmer, supports a different kind of focusing: swiping over the code to filter data based on the selected statements. But here again, the focusing only shows the data from executions on test input, which again might not work well for incomplete loops.

The loop seeds in our work are related to the idea of "overrides" from Example-Centric Programming [5] and "overwrites" from REPLugger [4], both of which allow the programmer to provide hypothetical values for certain variables. However, the prior work on REPLugger and Example-Centric Programming does not identify the important connection to loops and the loop-datavoid problem. Indeed we show that loop seeds are not just useful, but in some cases essential in a fundamental way, in that without them some loops just exhibit no live data when they are being written.

Finally, our work is related to *expression focusing* from Sketch-n-Sketch [9], which is a direct manipulation editor that translates changes made to the visual output of a program back to the source code. Expression focusing in this setting is a way of controlling the scope of the code that will be automatically modified when changing the visual output through direct manipulation. Although our work and expression focusing both have a notion of syntactic focusing, our work is in very different setting, and addresses a different problem altogether.

```
1  def sort(l):
2      res = []
3      for i in l:
4          added = False
5          for idx, j in enumerate(res):
6              if i < j:
7                  res.insert(idx,i)
8                  added = True
9                  break
10         if not added:
11             res += [i]
12     return res
```

Figure 1. Insertion Sort Full Code

```
1  def sort(l):
2      res = []
3      for i in l:
4          # insert i in res
5      return res
6
7  sort([5,4,8,1])
```

Figure 2. Partially Written Insertion Sort

PROBLEM OVERVIEW

A Task with Loops

Consider the task of writing *insertion sort*. Insertion sort works by maintaining a sorted result list that is iteratively populated with each element of the original list. There are many possible implementations of insertion sort. Figure 1 shows one such implementation using the Python programming language. This implementation was written by one of our human subjects.

To enable visualization of runtime values, let's assume that the programmer provides a sample input to `sort`, say the list `[5,4,8,1]`. While live coding visualizations can be useful once the code is written, the ultimate goal is to have such visualizations be helpful *while* the code is being written. With this in mind, consider the situation that occurs after the programmer has written only lines 1,2 and 3 in Figure 1. At this point, the programmer is trying to write the rest of the loop body, namely lines 4-11. In particular, the programmer is faced with the partial code as shown in Figure 2.

The Lack of Data

The problem with Figure 2 is that the data displayed by any live visualization at this point provides no assistance in writing the body of the loop. Indeed, running the partial code in Figure 2 on the input `[5,4,8,1]` will display `res` as the empty list on line 4. This is not useful because at this point the programmer is trying to write code to insert `i` into the sorted list `res`, and the empty list is a degenerate corner case for this code. Ideally, the programmer would be able to see `res` as a sorted non-empty list so that the programmer can get immediate feedback on finding the right place to insert into `res`.

We make the unavailability of good data in this example more concrete through the use of a live visualization called Projection Boxes [14]. Figure 3 shows the incomplete code from Figure 2, with Projection Boxes turned on. To make the example more salient, we added in Figure 3 the next step in writing the code, which is the inner "`for idx, j`" loop (note that all users in our study who took this approach wrote this loop first before realizing they need the variable `added`). Projection Boxes show at each line in the program the values of all

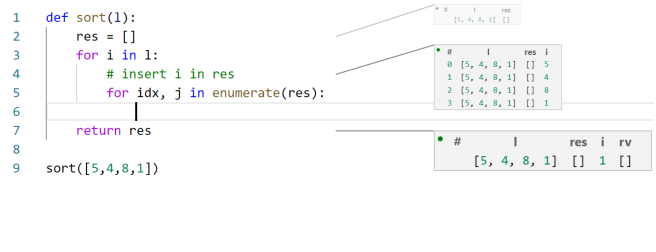

Figure 3. Partial Sort with Live Programming (but without FLiPS)

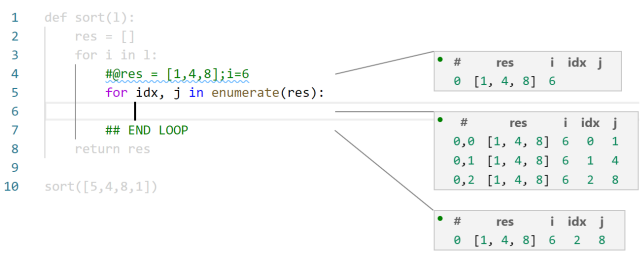

Figure 4. Partial Sort with FLiPS

variables at that line. Each box is a table of values and there is a straight line that connects the box to the place in the code that the box is visualizing. If the statement at a given line is executed multiple times, the box for that line contains one row for each execution of the statement. For example, the projection box at line 4 of Figure 3 has 4 rows because the loop is executed 4 times, with the value of `i` iterating through the values from the list `[5,4,8,1]`.

Using Projection Boxes, we can now see the problem that the programmer faces more concretely: in Figure 3 at line 4, the value for `res` displayed in the projection box is `[]` for all iterations of the loop. Even worse, since `res` being empty means that line 6 is never executed, *there is no projection box at line 6, which is the place where the programmer is about to write code*. Thus, as the programmer writes the code to insert `i` into `res`, there is no meaningful feedback from the live visualization. In fact, in our study, programmers in the control condition (who did not use FLiPS) were often puzzled as to why there was no data in the loops they were writing, sometimes thinking they had discovered a bug in the tool.

Loop-datavoid Problem

We call the above problem (namely the lack of live data while writing loops) the *loop-datavoid* problem. More concretely: *Given a loop which generates data that it also references, the loop-datavoid problem occurs when the live values inside the loop are either non-existent, incomplete or incorrect because the loop has not been fully written yet.* One important contribution of this paper is the identification of this problem. It has the potential to happen in any live environment for a imperative programming language that has loops.

The Problem Persists

The loop-datavoid problem is particularly problematic for two reasons.

First, there is no input that the user can provide which would rectify the situation. Indeed, no matter what input is provided to `sort` in Figure 3, the `res` variable will still be the empty list on line 4, meaning line 6 will still show no information.

Second, the lack of data can often persist for the entire time that the programmer writes the loop, because useful values of loop variables are only produced by the loop body itself, and this does not happen until the loop is written in full. Indeed, note that in Figure 1, even after the programmer has written lines 4-10 of the loop, `res` will *still* remain empty. It is only when line 11 (the last line of the loop!) is written that the data for the entire loop populates; but at this point the loop has already been written and so the data inside the loop, while possibly useful for debugging, is in no way helpful *while* writing the code.

OUR SOLUTION

Loop Seeds

To address the loop-datavoid problem, we introduce what we call *loop seeds*. Loop seeds are values that are used to start a hypothetical loop iteration, thus producing data for the programmer to visualize. In the above example, when faced with writing the loop in Figure 3, the programmer could provide a loop seed, for example `res = [2,5,8]`. This loop seed represents a hypothetical value of `res` at the start of a hypothetical loop iteration. To be useful, the value of `res` must represent a value that could actual occur in the final version of the code. For insertion sort, `res` should be sorted, since in the final code `res` will remain sorted throughout.

Focused Live Programming with Seeds

We reify loop seeds in an approach called *Focused Live Programming with Seeds* (FLiPS). FLiPS is a mode that the programmer can enter when writing a loop to focus on a particular iteration of the loop. When the programmer is done writing the loop, the programmer can exit that mode. FLiPS combines three key ideas:

- **Loop Seeds:** To use FLiPS in a given loop, the programmer must provide *loop seeds*, as described above.
- **Iteration Focus:** When the programmer enters FLiPS in a loop, the live data visualization is modified to only display the one iteration with the loop seeds (while showing all iterations of loops that are nested within the current loop).
- **Code Focus:** When the programmer enters FLiPS in a loop, some form of visual cue is used to make the code of the loop body stand out compared to the surrounding code.

Although variations of the above three ideas have appeared in previous settings (which we discuss more in "Related Work"), we are not aware of work that combines all three of the above ideas together in one coordinated setting to address the problem of loops for live programming.

IMPLEMENTATION

Figure 4 shows our implementation of FLiPS on top of projection boxes, using the same example as before. To visually focus attention on the loop, our implementation grays out the code that is not in the loop. Line 4 uses a special syntax to provide loop seeds, in this case seeds for the loop body on lines 4-7 (the loop itself starts on line 3).

Let's look at the inner loop written at line 5 (which is inside the loop body that FLiPS is focusing on). With FLiPS, this inner loop now iterates three times because it goes through the elements of `[1,4,8]`, which is the loop seed for `res`. As a result, using FLiPS (Figure 4) there is a projection box at line 6 that provides helpful immediate feedback on how this inner loop works. Contrast this to the situation without FLiPS (Figure 3), where this inner loop did even execute once because the *actual* value for `res` is `[]`, and so no projection box was displayed at line 6.

In our implementation, programmers provide loop seeds using the syntax "`#@`", followed by a semicolon-separated list of assignments. Python comments start with "`#`" so this is just a stylized comment. This allows us to leverage the visual cues of comment syntax highlighting to communicate to the programmer that loop seeds are an auxiliary part of the code (with the squiggly underneath to show which loop seed we are focusing on). Furthermore, because semicolon in Python combines multiple statements in a single line, the sequence of characters after "`#@`" is in fact an executable Python statement. Not only does this make the implementation of seeds simpler, but more importantly, we have found in preliminary trials that explaining loop seeds as an executable statement is the best way for programmers to understand how they work.

In summary, FLiPS adds to projection boxes: the ability to provide loop seeds, the ability to focus on a single iteration based the loop seeds, and the grayed-out code.

FLIPS WORKFLOW

The workflow for using FLiPS is as follows.

Before FLiPS: The programmer first starts writing a loop within a live programming environment, in our case Projection Boxes. At the point where they begin writing the loop body, or at some point while writing the loop body, the programmer might realize that they are not satisfied with the live data they are getting. At this point, the programmer can decide to provide loop seeds. As soon as the programmer types "`#@`" or edits a line that starts with "`#@`", the programming environment switches into FLiPS mode.

During FLiPS: Once in FLiPS mode, the programmer can focus on a single iteration of the loop, seeing data that can assist in writing the loop body. Once the programmer has written the loop body, they can adjust the loop seeds to essentially test the loop body on different inputs, inspecting the values at all points in the loop body. This provides the very useful ability to test the loop body on as a piece of code that is essentially not part of the loop.

After FLiPS: Once the programmer is happy with their loop body, they can press "ESC" to escape out of FLiPS mode, at which point the programmer is back to seeing all loop iterations. When outside of FLiPS mode, there is a keyboard shortcut that programmers can use to switch back into FLiPS mode to further debug, test or understand their loop.

Question	Avg
1: I found FLiPS helpful	4.5
2: FLiPS helped me get more useful live data in loops	4.6
3: I found FLiPS easy to use	4.6
4: I would like to have FLiPS available	4.6
5: I think FLiPS will be useful beyond today's tasks	4.4
6: Utility of providing loop seed values	4.9
7: Utility of seeing one iteration of the loop at a time	3.9
8: Utility of graying out code that is not in the loop	3.5

Figure 5. Questions in survey along with average scores. Questions 1-5 are on a 5 point Likert scale with 1 being "Disagree" and 5 being "Agree". Questions 6-8 asked the users to rate the utility of three features of FLiPS, on a 5 point Likert scale with 1 being "Not Useful" and 5 being "Useful". All questions had a median of 5, except for Question 7 with median 4 and Question 8 with median 3.5.

EXPERIMENTAL STUDY

We ran a user study to understand: (1) whether FLiPS is helpful, and if so under what conditions (2) to what extent FLiPS helps programmers get live data while writing loops when compared to not using FLiPS (3) how programmers use FLiPS (4) what aspects of FLiPS are most helpful.

We recruited 10 programmers (8 men, 2 women) with between 2 and 15 years of programming experience, who rated their familiarity with Python as between 3 and 5 out of 5. None of the programmers had ever seen or used Projection Boxes.

We had four programming tasks: (A) insertion sort (B) dictionary manipulation that involves repeatedly inverting a dictionary (C) list manipulation where elements are removed if they don't make a palindrome with any prior elements in the list (D) list encoding that involves list rotation.

Each subject solved: two problems with projection boxes without FLiPS (control condition), and two problems with projection boxes with FLiPS (test condition). We used two orders of the problems: 5 subjects did problems A,B without FLiPS, then C,D with FLiPS; and 5 did problems C,D without FLiPS, then A,B with FLiPS. Thus, for each of the four problems, we had 5 subjects who did it with FLiPS and 5 who did it without.

We instructed participants (both FLiPS and control) to try to get live data when writing their loops. After the study we had subjects fill out a survey and conducted post-study interviews to further understand their experience.

SURVEY RESULTS

Figure 5 shows the questions that we asked participants, and the average score for each. We can see that overall: (1) subjects found FLiPS helpful, specifically in getting more useful live data in loops (2) subjects found FLiPS easy to use (3) subjects would like to have FLiPS available when using projection boxes (4) subjects felt FLiPS would be useful beyond the tasks they saw in the study (5) the most useful aspect of FLiPS *by far* is the ability to provide loop seeds, followed by the ability to display one iteration at a time, followed by the graying out

the code not in the loop. Although subjects found the graying markedly less useful (Question 8), in preliminary trials where we did *not* gray out the code, subjects were confused about how loop seeds worked, and specifically suggested that it would be helpful to have a visual mechanism to show that FLiPS is focusing on just the loop body.

EFFECTIVENESS AT GETTING USEFUL DATA IN LOOPS

We would like to quantify the effectiveness of FLiPS at enabling useful data in loops, where useful data means having a projection box with a non-empty list or dictionary to operate on. To this end, we compare the availability of useful data in loops when problems were solved in the control condition (without FLiPS) vs in the test condition (with FLiPS available). All participants were told to try to get as much useful data as possible when they are coding their loops.

Recall that we had 4 problems, and each problem was solved by 5 subjects without FLiPS and 5 subjects with FLiPS available. So we had a total 20 problems solved without FLiPS and 20 problems solved with FLiPS available.

Let's consider the control condition, which is without FLiPS. In only 3 out of the 20 problems without FLiPS did subjects have useful data while writing their loop. In these 3 cases, the subjects didn't explicitly try to have data – they just *unintentionally* wrote the code in a way that the list or dictionary was populated first. This could happen for example in Figure 1 if the programmer had written lines 4, 10 and 11 first, only then to write lines 5-9. In the 17 cases that did not have useful data, most didn't even show a projection box in the loop, as in line 6 of Figure 3, because the code was never executed. Of the 17 without useful data, in 4 cases the subject tried to change their code to get better data, but failed and gave up. Finally, in the 17 who did not have data, almost every single one had a bug they only discovered later, which they could have discovered earlier had they had useful data while writing the loop.

Now, for the test condition, where FLiPS was available, in 18 out of 20 problems, the subject realized they did not have useful data, and then used FLiPS to get useful data while writing the loop. Furthermore, there was 1 case where the subject did *not* use FLiPS but still had useful data because this programmer just wrote the code in a way that data was available. This gives us the following result:

> *In the control condition (without FLiPS) only 3 out of 20 had meaningful data while writing their loops, and in the test condition (with FLiPS available), 19 out of 20 had meaningful data (18 of which were enabled by FLiPS)*

The one remaining case in the test condition was a case where the subject did not have useful data, but did not use FLiPS; the programmer in this case wrote the code very quickly (and correctly), and later observed they just didn't need the immediate feedback, so they didn't bother with enabling FLiPS.

DISCUSSION AND LESSONS LEARNED

Applicability

Based on observations of users working with FLiPS and our own exploration of FLiPS, we have developed an understanding of what loops FLiPS is useful for. FLiPS is most useful for loops that have at least one *loop-carried* variable, which means a variable that directly or indirectly depends on its own value from some prior iterations (`res` in Figure 1). Furthermore, this loop-carried variable should have two properties.

First, the starting value for this loop-carried variable must be a corner case for the loop body, so that when the loop body is incomplete, this starting value provides little feedback for writing the loop. Second, the loop body must perform some non-trivial operations on this loop-carried variable. If the operations are too simple, the lack of data in the live visualization will not hinder the programmer much.

This can be summarized as follows:

> *FLiPS is most useful for loops that have at least one loop-carried variable with the following two properties: (1) the starting value for the loop-carried variable is a corner case (2) the loop body performs non-trivial operations on this loop-carried variable.*

In prior studies involving live programming [22, 13, 10, 14], out of 14 tasks, only one really required a loop with a non-trivial loop-carried dependency (Dijkstra's algorithm). This is because prior tasks are mostly ones that create a new data structure from a given one, as opposed to a computation that iteratively builds a data structure – this is also the difference between "map" operations and "reduce/fold" operations in functional programming or MapReduce. This points to the fact that the kinds of loop-carried dependencies we are discussing here may have been under-explored in the live programming literature.

Note that when using recursion instead of loops, the loop-datavoid problem will not occur if we assume that users provide sample inputs for each function: the sample inputs for a recursive function would essentially act as seeds.

How Programmers use FLiPS

Many programmers in the study used FLiPS for the purpose we actually designed it for, which is to have data *while writing a loop*. However, we also found that FLiPS has several other important use cases.

Reduce Information Overload. One big challenge with always-on live visualizations is *information overload*, which happens when the amount of displayed information becomes intrusive, overwhelming and/or distracting. Some users mentioned that by focusing on a single loop iteration, FLiPS in fact makes the live information more manageable. For example, subject 4 specifically mentioned that one situation they envision FLiPS being useful is a deeply nested loop that is executed many times, where without FLiPS mode there would be a large number of iterations, but with FLiPS mode, the programmer could focus on just a particular iteration of the most inner loop.

Testing and debugging while using FLiPS. Right after finishing to write a loop body, the loop seeds in FLiPS allow the programmer to test their loop body one input at a time, before they consider how the entire loop works. This is very useful usage modality for FLiPS – every programmer who used FLiPS did this. Subjects 4 and 6 mentioned specifically how useful this

can be, because in many cases it's hard to understand what inputs to provide to a function to create a particular input for the loop body (especially if the loop body is deeply nested inside other loops).

Debugging after leaving FLiPS. We also noticed that FLiPS can be useful for debugging a loop *after the loop body has been written and the programmer is no longer in FLiPS*. Indeed, suppose that the programmer has finished writing a loop (either with or without FLiPS) and is now out of FLiPS mode. They now find that there is a bug in the loop body. While a buggy loop body often *does* generate live data for all loop iterations, the generated data is difficult-to-predict, and worse yet sometimes violates intended invariants. This makes it difficult for the programmer to reason about the loop body and how it operates on the live data. Just as one example, consider insertion sort again: a buggy loop can generate intermediate iterations where the list is not even sorted. This makes it hard to debug the loop body, which was designed specifically with the assumption that the list will be sorted. In these kinds of situations, FLiPS allows the programmer to debug the loop body by seeing how it operates on a chosen seed for which they know precisely what *should* happen. Subject 7 and 9 not only used this debugging strategy during the study (entering FLiPS mode again after the entire loop was written), but they also mentioned in a post-study interview that this would be very useful in practice.

Code Understanding. Subject 5 said that FLiPS would be useful in understanding natural language processing code that they were writing. That code often has loops with loop-carried dependencies (usually on previous states earlier in the sentence), and subject 5 said it would be useful to try hypothetical iterations to understand how the model works.

All of the above uses can be summarized as follows:

> *In addition to being useful for getting better data when writing a loop, FLiPS is also useful for: (1) reducing information overload (2) testing and debugging and (3) code understanding.*

Situations that make FLiPS less useful

Our study also taught us where FLiPS would be less useful. One such situations was brought up by subject 2, who rated FLiPS 3/5 on Question 1 from Figure 5 and 2/5 in Question 5 (these were the single lowest scores). Subject 2 gave these low scores because in practice he felt that he would have written the body of the loop in a separate function, and added test cases to that function. This is indeed an important consideration:

> *One alternative to FLiPS is moving the loop body into a separate function.*

Subject 3 also made this observation, although both subject 2 and 3 still gave 4/5 to Question 4 ("Do you want to have FLiPS available"). This shows that because FLiPS needs to be explicitly invoked, it can still be considered useful as an available feature, even if invoked infrequently.

Finally, this brings us to our final observation. Previous work on Projection Boxes [14] showed that programmers have *very different* preferences when it comes to live visualizations. Some programmers like a lot of feedback, others want much less feedback. Some programmers want feedback frequently, some only want the feedback after they have written their code. We noticed that all these preferences affect the extent to which users will find FLiPS useful:

> *Preferences about the extent to which a programmer might look at a live visualization carry over to FLiPS.*

In fact, subject 2 above, who gave the lowest scores, mentioned that he did not look at the projection boxes much, and would have preferred they be turned off until he found a bug in his code. Projection Boxes do support this feature (including many other customizations), although in this study we did not focus on customizations of Projection Boxes, so it was not explicitly explained to subjects. However, the above observation does point to the fact that all the customizability of Projection Boxes will also be useful for programmers who are using FLiPS.

Limitations and Future Work

Our user study was a small-scale lab study with specific predefined tasks. Furthermore, because users were told about FLiPS right before the study, they might be more likely to try it. As such, it remains an open research question how our results generalize to programming in the wild, including how often the loop-datavoid problem manifests itself in practice. A more realistic study would help address these questions.

The mechanism of using comments to store loop seeds is ad-hoc, and could lead to several problems, for example loop seeds being overlooked or going out of date. An interesting future direction would be to automatically generate loop seeds from the program and the runtime state.

Another interesting research direction would be to explore how FLiPS (and Projection Boxes) could be made to work with more complex data, using abstraction or summarization.

Finally, as loop seeds can be seen as a form of "overrides" [5] and "overwrites" [4] that enable FLiPS, we believe that through further research, one could come up with a uniform framework for all kinds of input values: program input, function/method input, loop body input, and overrides/overwrites.

CONCLUSION

In this paper, we identified the loop-datavoid problem: in a live programming environment, when programmers write certain kinds of loops, there is no useful live data. This is because the data inside the loop is generated by the loop itself, and until the loop is fully written the data inside the loop is either incomplete or incorrect. To address this problem, we proposed the idea of *loop seeds*, which provide hypothetical values for loop variables. We reified loop seeds in a programming mode called Focused Live Programming with Seeds (FLiPS), which allows the programmer to focus on a hypothetical loop iteration. Through a user study, we showed that FLiPS is helpful and easy to use, and that it increases the availability of live data while writing loops. Finally, we also observed that FLiPS provides additional benefits, namely: (1) reducing information overload, (2) testing and debugging loop bodies one iteration at a time and (3) code understanding.

REFERENCES

[1] 2019. Alfie. https://alfie.prodo.ai/. (2019). Accessed: 2019-09-01.

[2] 2019. LightTable. http://lighttable.com/. (2019). Accessed: 2019-09-01.

[3] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. DOI: http://dx.doi.org/10.1109/ICSM.2015.7332518

[4] Glen Chiacchieri. 2018. REPLugger: a pleasant and scalable live coding editor. In *International Workshop on Live Programming Worskhop (LIVE 2018)*.

[5] Jonathan Edwards. 2004. Example Centric Programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04 Onward)*. Association for Computing Machinery, New York, NY, USA, 124. DOI: http://dx.doi.org/10.1145/1028664.1028713

[6] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368

[7] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0805688.

[8] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 379–390. DOI: http://dx.doi.org/10.1145/2984511.2984575

[9] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA.

[10] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 737–745. DOI: http://dx.doi.org/10.1145/3126594.3126632

[11] Saketh Kasibatla and Alessandro Warth. 2017. Seymour: Live Programming for the Classroom. In *International Workshop on Live Programming Worskhop (LIVE 2017)*.

[12] Saketh Ram Kasibatla. 2018. *Seymour: A Live Programming Environment for the Classroom*. Master's thesis. University of California, Los Angeles.

[13] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. 2014. How live coding affects developers' coding behavior. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2014), 5–8.

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

[15] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. DOI: http://dx.doi.org/10.1145/3276497

[16] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. DOI: http://dx.doi.org/10.1145/3290327

[17] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019).

[18] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming As a Use Case for Context-oriented Programming. In *Proceedings of the Workshop on Context-oriented Programming (COP '19)*. ACM, New York, NY, USA, 17–23. DOI: http://dx.doi.org/10.1145/3340671.3343358

[19] S. L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

[20] Bret Victor. 2012a. Inventing on Principle. (2012). https://vimeo.com/36579366#t=18m05s

[21] Bret Victor. 2012b. Learnable Programming. (2012). http://worrydream.com/LearnableProgramming/

[22] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 258–265. DOI: http://dx.doi.org/10.1145/258549.258721