Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments

Ryo Suzuki
University of Colorado Boulder
ryo.suzuki@colorado.edu

Andrew Head
UC Berkeley
andrewhead@berkeley.edu

Gustavo Soares
UC Berkeley, UFCG, Brazil
gsoares@dsc.ufcg.edu.br

Loris D'Antoni
University of Wisconsin-Madison
loris@cs.wisc.edu

Elena Glassman
UC Berkeley
eglassman@berkeley.edu

Björn Hartmann
UC Berkeley
bjoern@eecs.berkeley.edu

Paste the appropriate copyright statement here. ACM now supports three different copyright statements:

- ACM copyright: ACM holds the copyright on the work. This is the historical approach.
- License: The author(s) retain copyright, but ACM receives an exclusive publication license.
- Open Access: The author(s) wish to pay for the work to be open access. The additional fee must be paid to ACM.

This text field is large enough to hold the appropriate release statement assuming it is single spaced in a sans-serif 7 point font.
Every submission will be assigned their own unique DOI string to be included here.

Abstract

In large programming classes, providing feedback on student coding mistakes during homework assignments is challenging. Recent advances in program synthesis have enabled automatic code hints that expose the location of the mistakes or suggest the required change to fix them. However, these hints may reveal too much information about the correct solution, which may affect student's learning outcomes, and they also lack information on why the code is wrong. To understand how the teaching staff gives feedback, we interviewed one teaching assistant and studied 132 Q&A interactions in the online discussion forum of an introductory computer science class. We found that the teachers often abstract the suggestion without revealing the answer and give targeted questions that suggest where they should focus on in the debugging process. Based on this study, we design five types of hints that transform a synthesized program into different information to explain mistakes, such differences in the program state and example of correct API usage. As future work, we will deploy and evaluate our hint system on real coding mistakes made by students in an introductory programming class at UC Berkeley.

Author Keywords

computer science education; automated feedback; program synthesis

```
def accumulate(combiner, base, n, term):
    total = 0
    if n==0:
        return combiner(base, 0)
    else:
        while n>0:
            total = combiner(term(n), total)
            n -= 1
    return total
```

Input	Expected	Actual
accumulate(mul, 2, 3, square)	72	0

Figure 1: An example of a student mistake on a programming assignment. The assignment asks students to write an `accumulate` function that returns the result of combining the first n terms in a sequence, e.g. `accumulate(mul, 2, 3, square)` $= 2 * 1^2 * 2^2 * 3^2$. The figure in the bottom shows an example of test-based feedback that suggests `accumulate(mul, 2, 3, square)` should return 72 instead of 0.

```
  def accumulate(combiner, base, n, term):
-     total = 0
+     total = base
      if n==0:
          return combiner(base, 0)
      else:
          while n>0:
              total = combiner(term(n), total)
              n -= 1
      return total
```

Figure 2: An example of the synthesized code fix for the student mistake. This fix is generated by using Refazer, an existing program transformation system [21].

ACM Classification Keywords

H.5.m [Information interfaces and presentation (e.g., HCI)]: Miscellaneous

Introduction

Feedback is crucial to learning programming [2, 17, 22]. Personalized, timely feedback can help students get unstuck and correct their misconceptions [6, 12]. However, such personalized attention does not scale in large computer science classes [7, 9]. Many large introductory CS courses address this scalability problem by using automated grading and feedback tools [15]. For example, MIT and UC Berkeley use automatic testing tools that report failed results based on teacher-written test cases [3]. However, test-based feedback may leave students with a large gulf of evaluation [19] on how to map the failed test results to the error in their code [13].

Recent advances in program synthesis have enabled to produce more specific hint to fix the student mistake [14, 20, 21, 23]. These systems can automatically synthesize a code fix and then translate the synthesized information into a bottom-out hint, which directly suggests the required changes to fix their code [20, 23]. For example, Fig. 1 and 2 illustrate an example of a student mistake and synthesized code fix generated by Refazer, an existing program transformation technique [21]. The current feedback system such as AutoGrader [23] then would turn this information into a bottom-out hint like *"In the expression total = 0 in line 2, replace the value 0 to base"*.

However, the bottom-out hint exposes a direct solution, which would lead students to fix their code without understanding the underlying misconception [25]. Moreover, the current hint focuses on *how* to fix the code, but does not address *why* the current code is wrong. In contrast, a human teacher would feedback without giving away a solution or give scaffolds to understand the problem. For example, a teacher would feedback on the mistake in Fig. 1 by abstracting the suggestion like *"make sure to initialize with the correct value"* or by illustrating the execution flaw to explain why the code returns 0 instead of 72. Such well-designed feedback can effectively facilitate students to understand their problems [25].

In this paper, we explore the design space of automatically synthesized hints for introductory programming assignments. Our goal is to address the design problem of how to produce pedagogically appropriate feedback given the synthesized program transformation. To understand the design guideline and strategies that TAs employ, we reviewed 132 Q&A Piazza posts and interviewed one teaching assistant of the introductory computer science class at UC Berkeley. The study shows that teachers often ask scaffolding questions like *"What happens when you call accumulate functions?"* to give students a conceptual guidance where they should focus on in the debugging process. Also, the teachers avoid revealing a direct path to a solution. Instead, they often abstract the suggestion with a high-level hint like *"check the number of the argument in the function"* or *"missing the base case in the recursive function"*.

We then identify the following five different types of hints based on the strategies that TAs employed on Piazza posts; (1) **Location hint**: to help students locate the error, (2) **Data hint**: to illustrate why the code fails with a specific example, (3) **Behavioral hint**: to provide a higher-level hint of the execution behavior of the code, (4) **Transformation hint**: to give a hint how to fix the code, and (5) **Example-based hint**: to give an example that illustrates how to use a function. We implement a prototype system that shows a subset of these hints and develop a user interface that al-

lows student interactively examine the code mistakes with the suggested hints.

We contribute:
1) an analysis of the design space of realizable hints from automatic analysis of student submissions to programming assignments using state-of-the-art synthesis techniques;
2) the design of a concrete feedback system that implements a subset of the hints identified above for an introductory computer science class;

Related Work

Automated Feedback for Programming Assignment
Most common approach to providing scalable feedback is a test-based feedback which reports the failed test results against teacher-written or algorithmically generated test cases [24, 4]. However, novice learners find it difficult to map the failed test results to the error in their code [13, 23]. Recent advances in program synthesis have demonstrated an alternative approach to providing more targeted code hints. AutoGrader [23] leverages program synthesis techniques to find the bug-fix repair, and then automatically generate hints based on the program repair. While AutoGrader uses pre-defined error models to find bug-fixes, the example-based methods address this limitation by learning syntactic code transformations from the previous student submissions [21]. Rivers et al. [20] also show the data-driven approach can cover a larger solution space compared to the existing program synthesis approach. These techniques can be used not only for generating automatic code hints, but also for detecting and clustering the common misconception and similar errors [9, 13].

The current systems turn the synthesized information into a bottom-out hint [20, 23]. However, these hints may reveal too much information about the correct solution, which may allow students fix their code without the understanding of the problem, or much worse game the system [1, 28]. Furthermore, the hints lack a pedagogical information to help students understand *why* the code is wrong. For novice learners, understanding the behavior of the program is the most important aspect of debugging process [8, 16, 12].

Principles for Feedback Design
Prior work draws four essential elements that the debugging assistants should provide [18]; (1) help students locate the bug [22, 26]; (2) demonstrate an instance in which the code fails [5, 26]; (3) explain the behavior of code with a visual execution of the code [5, 11, 10]; (4) help students comprehend the relation between the symptoms and the cause of the error [17, 16]. Based on these pedagogical principles for debugging assistance, this paper focuses on the design challenge on how to turn the program synthesis information into appropriate feedback. Recent work has explored the similar design challenge of automatically generated hints for the restricted domain of finite automata [7]. Taking inspiration from these work, we generalize this approach to more complex domain of introductory programming assignments.

Design Goals

To understand how teaching assistants currently give feedback on student mistakes, we conducted a formative study of programming assignments in an introductory computer science class (UC Berkeley CS61A) We first reviewed 132 posts on Piazza discussion forum to investigate what the students ask about and how the teachers respond. Then, we also had a semi-structured interview with a teaching assistant of the same course. We summarize the result of the study as the following design goals of the effective feedback from the pedagogical point of view.

Facilitate understanding of "why" the code is wrong.
Typical automated feedback like test-case results or point hints can provide high-level hints to help students understand why the code is wrong. However, the high-level feedback about why the code is wrong (test case feedback) is sometime not enough. In fact, we observe many students still asked TAs to help understand "why" the code is wrong. For example, *"Q: I don't understand why the system returns the wrong value (15) for my code [...]."* [id: 100] These questions include *"why the value differs from the expected one?"* [id: 63, 100], *"why not working for the specific test case?"* [id: 64, 80, 112, 117], *"why the code gets a syntax or runtime error?"* [id: 56, 86, 91], *"why the code behaves in a weird way?"* [id: 84, 129].

In response to such questions, TAs often recommended students to look into the execution flow with PythonTutor [PythonTutor], an interactive code execution visualizer tool. Among the 132 total responses, the TAs mentioned about PythonTutor in 19 times. For example, *"A: put your code in python tutor and see where it goes wrong, that is a way to help you debug and learn the debugging process."* [id: 100]

Although PythonTutor can effectively visualize the code execution flow, students are sometimes overwhelmed by too much information and the lack of focus. For example, some students still asked a question about how to interpret the result [id: 38, 41]. Therefore, the teaching assistants give students scaffolds to draw the attention, targeting questions to suggest where the student should focus on. *"Try to examine the code in PythonTutor. What happens when you call accumulate? Is the combiner that you're passing on to accumulate making a decision based on the predicate for every number in the sequence?"* [id: 74]

This insight lead us to formulate the following design goals: (1) helping to understand "why" the code is wrong, not only with respect to test cases but also in terms of each step of the program execution, as in a debugging process, (2) guiding students where they should focus on in the debugging process.

Abstract the suggestion
The program synthesis based feedback can give low-level hints as bottom-out hints (*e.g. change the value of the variable i from 0 to 1 in line 5*). However, one of TAs we had interviewed concerned that it can potentially cut off the learning opportunity. *"TA: We don't want to give away of the solution because it cuts off the learning opportunity. Students also do not like to have just an answer. So, instead, we try to give conceptual guide like Have you thought about X? or What happen if X?"* [Interview]

Our study of the interaction in Piazza also supports this principle. None of the TAs revealed the direct path to the solution in the responses to student questions. Instead, they abstract the suggestion and give a conceptual guidance. *"Q: "Runtime Error - Maximum recursion depth exceeded in comparison" message shows up. What is the meaning of this error message?" "A: It means that you do not have a base case that can stop the program from running your recursive calls."* [id: 60]. TAs guide students by abstracting *"check the number of the argument in the function"* [id: 90, 98], *'you are making too many recursive calls"* [id: 17], *"check the variable of x and y"* [id:84], *"make sure the parentheses to be closed before the line"* [id: 10], *"check the if statement"* [id: 86], *"missing the base case in the recursive function"* [id: 60, 117]. Therefore, this motivates us to design the hint as abstracting the suggestion without revealing the solution.

Design Space of Automated Hints

Based on our literature review and formative study, we design five types of hints that can be generated with the synthesized program repair. These hints are designed to target different aspects of students' debugging activities: (1) locate the error in the code; (2) comprehend the code with a concrete example (3) understand the behavior of the code execution (4) relate the error with underlying misconceptions

Location Hint: The location hint provides information about which part of the student code is incorrect. For instance, a location hint for our running example would be: *"There is an error in line 3"*. The level of abstraction of a location hint can vary. A more concrete hint would be: *"There is an error in the value assigned to the variable total in line 3"*. This type of information is easily extracted from a synthesized bug fix. However, there may be different locations where the student code could be fixed. For instance, an off-by-one error in a loop can be fixed by changing the initial value of the iterator or changing the stop condition. If the synthesized fix changes the program in an unexpected way, the student may have problems understanding why the pointed location is incorrect.

Data hint: Only identifying the location of the bug may not be enough. The most important step in the debugging process is to understand the behavior of the program [8]. However, novice learners often struggle to comprehend the behavior of the code without concrete examples of the data [16, 27]. Data hint provides information about expected internal data values of the program during a debugging section. The system iteratively executes the code, line-by-line, similar to a debugging tool such as PythonTutor [11]. When the system detects that a value of variable is incorrect, it pauses the execution of the program, and shows the difference between the expected value and the actual value.

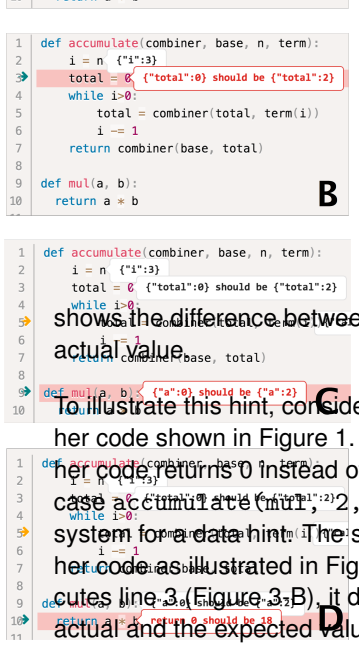

Figure 3: An example of the hint interface of data hint with the same mistake illustrated in Figure 1 and 2. The system visualizes how the student mistake, "total = 0" instead of "total = base", affects the difference in the result.

To illustrate this hint, consider a student who is trying to fix her code shown in Figure 1. Julie does not understand why her code returns 0 instead of 72 when executed on the test case `accumulate(mul, 2, 3, square)`. Julie asks the system for a data hint. The system starts the execution of her code as illustrated in Figure 3. When the system executes line 3 (Figure 3.B), it detects a difference between the actual and the expected values. The system pauses the execution and shows a message saying that the actual value of the variable `total` should be `2` instead of `0`. Now, Julie understands why her code is returning `0`: the initial value for the multiplication is `0`. By looking at the expected value (2) she also realizes that the initial value of the accumulate operation, the variable `total`, should be the `base` parameter. After performing this change, her code passes all test cases.

Behavioral hint: The Data hint points to a difference in the program state in a particular moment of the program execution. Sometimes, however, it is hard to understand how that difference affects the program behavior considering the entire program execution. The Behavioral hint provides information about the internal behavior of the program during its execution. For instance, in our running example, a Behavioral hint would be: "The variable total received the following values after the program execution: 0, 0, 0, 0. The expected values are: 2, 18, 72, 72. As another example of a Behavioral hint, consider a scenario where the student had forgotten to add a base case to the recursion, and the code thrown a `StackOverFlowException`. A Behavioral hint will show the difference in the expected number of calls to the recursive function and the actual one.

Transformation hint: Sometimes students may locate

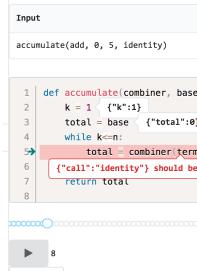

Figure 4: The user
current hint system.
clicks the button in t
system shows the tr
line-by-line code exe
animation. The syst
animation when the
diverges, then highli
with the difference i
data value or a beha

the part of the code that is wrong and understand why it is wrong but do not know how to fix it. The Transformation hint abstracts the synthesized fix and provides an information about what change the student should do. For example: *"you should replace the initialization of the variable total"* or *"you should add an If statement"*. The abstraction level can be configured by the teaching staff. This type of hint is already used by current program synthesis techniques.

Example-based hint: Another way of showing how to fix the code is using a similar example. In our study, we found that the teaching assistants often used this type of hint. For instance, in one scenario, the student did not know how to use the `combine` function. The TA explained to him: *" accumulate(add, 0, 5, identity) should return 0 + 1 + 2 + 3 + 4 + 5. In this case combiner is the two-argument add function, which we use like this: 0 + 1 + 2 + 3 + 4 + 5 = ((((0 + 1) + 2) + 3) + 4) + 5 = add(add(add(add(add(0, 1), 2), 3), 4), 5)."*. In another scenario, the TA gave an example of proper way of using lambda functions.

Current implementation
We implement a system that gives a subset of the hints proposed in the above, including the Location hint, the Data hint, and the Transformation hint. Our system is based on program transformations synthesized by Refazer [21], an existing data-driven program transformation technique. Given the student's incorrect submission, Refazer can synthesize a fixed code by transforming the buggy code. Using this pair of buggy and fixed code, the system traces the line-by-line execution for each code with failed test cases. For each step, the system stores the state of the execution, including a line number, a function call, a list of variables and data, and the differences from the previous step. By comparing the difference in the code execution, we can identify how and when the execution result diverges. Using this information, the system visualizes the trace of code execution within an interactive debugging interface shown in Fig. 4.

Future Work

In this paper, we explore the design space of automated hints for students in introductory programming courses. We present five types of automated hints and describe the implementation of three of them. As future work, we plan to implement the other two types of hints and evaluate them.

To implement Example-based hints, we plan to extract the list of Python constructs and APIs presented in the code changed by the synthesized fix. For each element in the list, we will search for examples of code snippets that contain the element on the Python documentation, Course documentation, and StackOverflow. We can implement the Behavioral hint using the same infrastructure of the Data hint. However, the internal behavior of the incorrect program may diverge of the correct one in many different ways, such as, variable values, number of function calls, covered branches. We plan to investigate how to identity the most valuable information for students and how to present it.

We also have not investigated how these hints should be delivered and combined. In other words, which hint is effective in what kind of situations. For example, location hint may sometimes be enough to correct the bug for a simple error, or showing all of hints can be distracting or misguiding. As future work, we will to deploy these hints in the introductory programming class of UC Berkeley. We plan to conduct an in-class user study to evaluate the effectiveness of these hints on students.

Acknowledgments

This research was supported by the NSF Expeditions in Computing award CCF 1138996, NSF CAREER award IIS 1149799, CAPES 8114/15-3, an NDSEG fellowship, a Google CS Capacity Award, and the Nakajima Foundation.

References

[1] Vincent Aleven, Bruce McLaren, Ido Roll, and Kenneth Koedinger. 2004. Toward tutoring help seeking. In *Proceedings of ITS (ITS '04)*. Springer, 227–239.

[2] Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. 2010. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons.

[3] Soumya Basu, Albert Wu, Brian Hou, and John DeNero. 2015. Problems before solutions: Automated problem clarification at scale. In *Proceedings of L@S (L@S '15)*. ACM, 205–213.

[4] Judith Bishop, R Nigel Horspool, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2015. Code Hunt: Experience with Coding Contests at Scale. In *Proceedings of ICSE (ICSE '15)*. IEEE, 398–407.

[5] Peter Brusilovsky. 1993. Program visualization as a debugging tool for novices. In *Proceeding of CHI (SIGCSE '93)*. ACM, 29–30.

[6] Albert T Corbett and John R Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of CHI (CHI '01)*. ACM, 245–252.

[7] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 9.

[8] David J Gilmore. 1991. Models of debugging. *Acta psychologica* 78, 1-3 (1991), 151–172.

[9] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 7.

[10] Mitchell Gordon and Philip J Guo. 2015. Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In *Proceedings of VL/HCC (VL/HCC '15)*. IEEE, 13–21.

[11] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of SIGCSE (SIGCSE '13)*. ACM, 579–584.

[12] Philip J Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of UIST (UIST '15)*. ACM, 599–608.

[13] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Bjoern Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of L@S (L@S '17)*. ACM.

[14] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. In *Proceedings of FSE (FSE '16)*. ACM.

[15] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises–Extended Version. (2016).

[16] Andrew Ko and Brad Myers. 2008. Debugging reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of ICSE (ICSE '08)*. IEEE, 301–310.

[17] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-user Programming Systems. In *Proceedings of VL/HCC (VL/HCC '04)*. IEEE, 199–206.

[18] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[19] Donald A Norman and Stephen W Draper. 1986. User centered system design. *New Perspectives on Human-Computer Interaction* (1986).

[20] Kelly Rivers and Kenneth R Koedinger. 2015. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* (2015), 1–28.

[21] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of ICSE (ICSE '17)*. IEEE.

[22] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.

[23] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of PLDI (PLDI '13)*. ACM, 15–26.

[24] Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *Proceedings of ICSE (ICSE '13)*. IEEE, 1117–1126.

[25] Kurt Vanlehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.

[26] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[27] Bret Victor. 2012. Learnable programming. *Worrydream. com* (2012).

[28] Jason A Walonoski and Neil T Heffernan. 2006. Detection and analysis of off-task gaming behavior in intelligent tutoring systems. In *Proceedings of ITS (ITS '06)*. Springer, 382–391.