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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
CHI'17 Extended Abstracts, May 06-11, 2017, Denver, CO, USA
ACM 978-1-4503-4656-6/17/05.
http://dx.doi.org/10.1145/3027063.3053187

Abstract

For massive programming classrooms, recent advances in program synthesis offer means to automatically grade and debug student submissions, and generate feedback at scale. A key challenge for synthesis-based autograders is how to design personalized feedback for students that is as effective as manual feedback given by teachers today. To understand the state of hint-giving practice, we analyzed 132 online Q&A posts and conducted a semi-structured interview with a teacher from a local massive programming class. We identified five types of teacher hints that can also be generated by program synthesis. These hints describe transformations, locations, data, behavior, and examples. We describe our implementation of three of these hint types. This work paves the way for future deployments of automatic, pedagogically-useful programming hints driven by program synthesis.

Author Keywords

programming education; automated feedback; program synthesis

ACM Classification Keywords

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

```
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 an incorrect student submission for a programming exercise in a massive programming class. The bottom widget shows typical feedback that a student sees, comparing the expected and actual, erroneous output of the student's program on a set of test inputs.

```
 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 [15].

Introduction

Personalized, timely feedback can help students get unstuck and correct their misconceptions [2, 8]. Teachers' personalized attention does not scale in massive programming classes [3, 5]. Instead, it is common for teachers to provide test case suites, which students can test their submissions against (Figure 1).

This substitution has some drawbacks. While a teacher might look at the student's submission and recommend reviewing a particularly relevant lesson or attempt to reteach an important concept, test case feedback can only point out how the student submission does not return the right answer. It can be difficult for a student to map failed test results back to a specific error in their code.

Recent advances in program synthesis provide personalized feedback at scale; this feedback helps students fix incorrect submissions to programming exercises [10, 14, 15, 17]. These systems use program synthesis to learn code transformations that fix incorrect student submissions. Fixes can be shown to students as *bottom-out hints* that explain the exact changes needed to fix the code [14, 17]. For example, the synthesized fix in Figure 2 could be mapped to feedback like *"In the expression* `total = 0` *in line 2, replace the value* `0` *with* `base`*."*.

Bottom-out hints alone can undermine the pedagogical value of students debugging why their code is wrong. Well-designed feedback helps students understand their problems and debug their own submissions [16, 18]. As we found in our formative study, teachers prefer to present higher-level feedback without giving away the solution.

A key challenge in automatic hint generation is providing feedback that facilitates productive debugging, rather than enabling a student to skip debugging altogether. In this paper, we explore a design space of hints that can be automatically generated from code transformations learned by program synthesis. Our ultimate goal is to adapt the strategies that a human teacher employs to automated hints driven by program synthesis.

To understand the differnt types of hints teachers manually give in programming classes, we analyzed 132 Q&A posts from a discussion forum and interviewed a teaching assistant from UC Berkeley's introductory programming class. We identified five types of hints that could be generated from synthesized code transformations: **transformation hints**, recommendations of abstract or concrete fixes to apply to incorrect code; **location hints**, pointers to code entities that need to be understood or fixed; **data hints**, the expected type or value of a variable at one point in a trace; **behavior hints**, descriptions of intended or abnormal dynamic program behavior; and **example hints**, clarifications of the values and types of input and output that a program must satisfy. We implemented prototypes for three of these hints, embedded in an interactive debugging interface, to help students answer clarifying questions about bugs in their programs.

In this work, we contribute: (1) a characterization of five types of hints that can be generated by state-of-the-art synthesis techniques, informed by a formative study; (2) the implementation of transformation, location, and data hints in an interactive debugging interface, appropriate for deployment and evaluation in a massive programming classroom.

Related Work

Automated Feedback for Programming Assignment

Intelligent tutoring systems (ITS) often supply a sequence of hints that descend from high-level pointers down to specific, bottom-out hints that spell out exactly how to generate

the correct solution. For example, in the Andes Physics Tutoring System, hints were delivered in a sequence: *pointing*, *teaching*, and *bottom-out* [19]. ITS have been historically expensive and time-consuming to build because they relied heavily on experts to construct hints.

Recently, researchers have demonstrated how program synthesis can generate some of those personalized and automatic feedback typically found in ITS's [10, 14, 15, 17]. For example, AutoGrader [17] can identify and fix a bug in an incorrect code submission, and then automatically generate sequences of increasingly specific hints about where the bug is and what students need to change to fix it.

High-level hints that point to relevant class materials or attempt to reteach an concept can be difficult to automatically generate because they require more context or the deep domain knowledge of a teacher. Recent work has demonstrated how program analysis and synthesis can be used as an aid for a teacher to scale feedback grounded in their deep domain knowledge [5, 9]. While scaling up teacher effort, these systems still require teachers to manually review and write hints for incorrect student work.

In contrast to that earlier work on scaling up teacher-written feedback, this paper focuses on *fully automated* ways to provide high-level hints, specifically for the context of writing code. D'Antoni et al. [3] has explored the similar design challenge of automatically generated hints for the domain of finite automata [3]. They propose two hints to explain *why* the student solution is wrong and one hint to explain *how* to fix the solution. Taking inspiration from this work, we design data and behavior hints to help students answer *why* question, and location and transformation hints to address *how* question in the domain of introductory Python programming assignments. While the simpler nature of the automata domain allowed authors to propose hints that describe in a high-level language what the student solution computes, in our domain, we compare the internal state of the incorrect submission to the fixed one to explain *why* the code is incorrect.

Principles for Feedback Design

Prior work describes the following four essential elements that debugging assistants should provide [13]: (1) help students locate the bug [16, 20]; (2) demonstrate an instance in which the code fails [1, 20]; (3) explain the behavior of code with a visual execution of the code [1, 7, 6]; (4) help students comprehend the relationship between the symptoms and the cause of the error [12, 11].

We design different types of hints that target different aspects of these elements: (1) location hints may help students locate errors, (2) data and example hints may help students comprehend code through failed test cases, and (3-4) behavior hints may help students understand dynamic behavior and map the relationship between the error and the cause.

Formative Study

To understand how teaching assistants provide hints, we conducted a formative study with the teaching staff of an introductory programming class (UC Berkeley CS61A). We first reviewed 132 Q&A posts in CS61A's online forum to investigate how teachers gave hints in response to student questions to one particular programming assignment. We then conducted a semi-structured interview with a teaching assistant from the same course.

The first author performed open coding on the 132 Q&A posts by reading each post and determining common themes in the structure and focus of teachers hints. Two authors reviewed the resulting hint types, composing a definition and hint example for each type. They independently performed

axial coding to tag each post with hint types it contained. Then the authors reviewed discrepancies in their analysis results, resolving differences by discussing them until reaching consensus. The analysis yielded ten types of hints; location, data, behavior, transformation, example, usage, references, diagnosis, PythonTutor, and diagram. Of the 132 posts, 70 contained at least one teacher response with a hint of one of these ten types. Of the ten types identified through coding Q&A posts, we chose five (transformation, location, data, behavior, and example hints) that are most amenable to program synthesis.

Hints answer "why" questions about failing code.
When debugging incorrect code, students in CS61A receive feedback from an autograder as a list of test cases that their code fails. However, this feedback was not always enough: students often asked in the class forum for help understanding why their code failed. Teachers would point out failed test cases (posts 64, 80, 112, 117), unexpected values (63, 100), and syntax and runtime errors (56, 86, 91).

In response to these "why" questions, teachers recommended that students learn more about the behavior of their programs. One common piece of feedback (appearing 19 times) was to run their code in PythonTutor [7], an interactive code visualization tool. However, students could be overwhelmed by too much information. Teaching assistants provided scaffolding by pointing to specific locations (14 times), expected data and types (5 times), and clarifications of runtime behavior (12 times), to help focus attention.

Hints rarely provide exact fixes
Recent program synthesis techniques can easily recommend concrete fixes to student code. However, teachers rarely hinted at exact fixes. Concrete fixes were typically only given for one-off syntax errors like missing parentheses (posts 56, 105, 127). In our semi-structured interview with a teacher from this course, they clarified, *"We don't want to give away 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 a conceptual guide like 'Have you thought about X?', or 'What happens if X?'"*

In addition to the four design principles discussed based on prior work, we identified following design goals to improve the potential of hints driven by program synthesis:

1. Only rarely provide bottom-out hints
2. Integrate hints with existing interactive debugging tools to encourage students to explore their code
3. Provide guidance through pointers to specific code locations and runtime behavior

Designing Program Synthesis-Driven Hints

Based on our review of related work and our formative study of teacher feedback, we designed five types of hints that can be generated with synthesized program repair. In this section, we introduce the hint types with examples of teacher feedback and describe an implementation strategy for each hint type. We categorized the teachers responses into five types of hints based on the following definitions: **transformation** that describes concrete or abstract fix that needs to be made, **location** that helps to locate the error by pointing to concrete or abstract location in code, **data** that describes incorrect state or type of data at a point in the trace, **behavior** that points out sequence of calls or data that represent behavior that needs to be changed, **example** that clarifies specification for program or sub-program by describing types/values of inputs/outputs of the function.

Our current implementation is based on Refazer [15], an existing program transformation system. As inputs, our system takes (1) student's incorrect code, (2) fixed code

Transformation Hint
Replace 0 with base in line 2

```
1  def accumulate(combiner, base, n, term):
2      total = 0
3      if n==0:
4          return combiner(base, 0)
5      else:
6          while n>0:
7              total = combiner(term(n), total)
8              n -= 1
9      return combiner(base, total)
```

Figure 3: Transformation Hints

Location Hint
Check the value of total in line 2

```
1  def accumulate(combiner, base, n, term):
2      total = 0
3      if n==0:
4          return combiner(base, 0)
5      else:
6          while n>0:
7              total = combiner(term(n), total)
8              n -= 1
9      return combiner(base, total)
```

Figure 4: Location Hints

Data Hint
Running accumulate(mul, 2, 3, square)
Expected 72 but got 0

```
1  def accumulate(combiner, base, n, term):  # acc
2      total = 0  # total = 0 should be total = 2
3      if n==0:
4          return combiner(base, 0)
5      else:
6          while n>0:
7              total = combiner(term(n), total)
8              n -= 1
9      return combiner(base, total)
10 
11 accumulate(mul, 2, 3, square)  # call: accumula
```

Figure 5: Data Hints

synthesized by Refazer, (3) transformation rules identified by Refazer, and (4) list of failed test results detected by an autograder.

Transformation Hints
Transformation hints provide information about concrete changes to code that can fix the current code:

"*In your `n == 1` base case, you should be calling `term(1)` and not `term(0)`. Remember that the `term` function is only defined from `1` to `n` inclusive.*" (post 82)

Refazer can generate abstract code transformations as a set of the abstract syntax tree (AST) operations. For example, an operation to fix the mistake in Figure 2 is *Update(ConstantExpressionNode, NameExpressionNode)*. Our system then turns these transformation rules into natural language by translating the abstract expressions (e.g. NameExpressionNode) into concrete expressions (e.g., base). Figure 3 shows an example transformation hint.

Location Hints
Currently, transformation hints generated by our system is equivalent to bottom-out hints. However, teachers may want to elide details of exactly where or how changes need to be applied (post 123). Location hints are one level more abstract than transformation hints. Location hints can be not only pointing out a position in code, but also focus a student's attention on a particular code structure or entity that needs attention:

"*Look carefully at your `else` case and also the condition of your `if`. Is it doing what you expect it to?*" (post 50)

The location that a student needs to fix in incorrect code can be detected by applying a synthesized transformation to an incorrect submission and marking which lines have changed (Figure 4). The level of abstraction of a location hint could vary from line number ("there is an error on line 3") to structure-aware ("take another look at your lambda").

Data Hints
While transformation and location hints point out where code is incorrect and how it can be changed, data and behavior hints help illuminate why a student's code fails. Data hints point out important values or types of variables:

"*Repeated returns a function. So, may I ask what are this returned function argument and return types.*" (post 27)

Novice learners often struggle to comprehend the behavior of the code without concrete examples of the data [11, 21]. Data hints provide information about expected internal data values of the program during a debugging session. Given the failed test case, these hints can be implemented by comparing the dynamic execution trace of an incorrect and fixed submission (Figure 5). When the system detects that a value of a variable in an incorrect submission diverges from that in the fixed submission, it can pause execution and highlight the difference between the expected value and the actual value.

Behavior Hints
One important step in the debugging process is to understand the behavior of the program [4]. In contrast to data hints which highlight a particular moment of the program execution, behavior hints describe the difference in sequences of program's state or control flow:

"*Think about what the counter value is, and what the total value is. Is this correct? Remember, ping pong looks like:*

```
total = 1 2 3 4 5 6 [7] 6 5
count = 1 2 3 4 5 6 7 8 9
```

for the first 9 elements." (post 28)

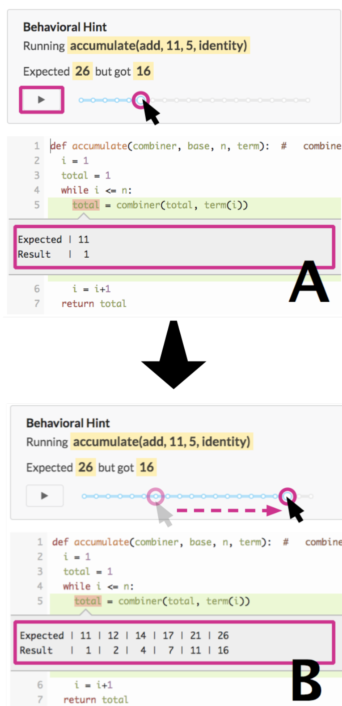

Figure 6: Behavior Hints

Although we acknowledge that the current implementation of behavior hints only works for a subset of student mistakes, we expect that behavior hints can be implemented using the same infrastructure as data hints. Our system stores the execution results as a sequence of the internal state. By comparing the difference in the sequences of the internal states, the system can highlight divergent behavior between the expected and current incorrect code, or as representations of control flow, both of which we observed in our formative study (Figure 6).

Example Hints
Example hints clarify the specification of a programming assignment by describing the expected input and output types and values for code constructs:

"*the identity function is a function that takes in some value x and just spits that value back out. For example, `identity(4)` returns 4*" (post 7)

We also observed that teachers provided examples of how to use lower-level constructs, such as Python lambdas and ternary operators. To implement example hints, we plan to extract the list of Python constructs and APIs presented in the code changed, then search for examples of code snippets from online resources or course documentation.

Discussion

There are three open questions. First, we have not evaluated how pedagogically useful these hints are. We plan to conduct an in-class user study to evaluate the quantitative (e.g., number of attempts or time spent to fix a bug) and qualitative (e.g., how useful a hint is for submitting correct code or understanding the student's initial mistake) value of these hints.

Second, we need to investigate how and when these hints should be shown to students. One standard practice is to show hints in sequence from high-level hints to bottom-out hints [19]. Such sequences can be also algorithmically determined based on tree edit distance [3]. Alternatively, mixed-initiative approaches might enable teachers to specify these sequences or level of abstraction without losing scalability [9]. We will explore both approaches and evaluate how these sequences affect learning outcomes.

Third, we are interested in investigating whether our system can provide useful hints beyond the introductory classroom. Our hints rely on the capabilities of program synthesis techniques to discover code transformations that fix incorrect code. While such techniques have been demonstrated on short assignments in introductory programming classrooms, in the future it may be possible to learn generalizable fixes for larger, more complex programs.

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] Kurt Vanlehn, Collin Lynch, Kay Schulze, Joel A. Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. 2005. The Andes Physics Tutoring System: Lessons Learned. *Int. J. Artif. Intell. Ed.* 15, 3 (Aug. 2005), 147–204.

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

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