Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis

Andrew Head[†]*, Elena Glassman[†]*, Gustavo Soares[†‡]*,
Ryo Suzuki[§], Lucas Figueredo[‡], Loris D'Antoni[‖], Björn Hartmann[†]
[†]UC Berkeley, [‡]UFCG, [§]CU Boulder, [‖]UW-Madison
{andrewhead,eglassman,bjoern}@berkeley.edu, {gsoares@dsc,lucas.figueredo@ccc}.ufcg.edu.br,
ryo.suzuki@colorado.edu, loris@cs.wisc.edu

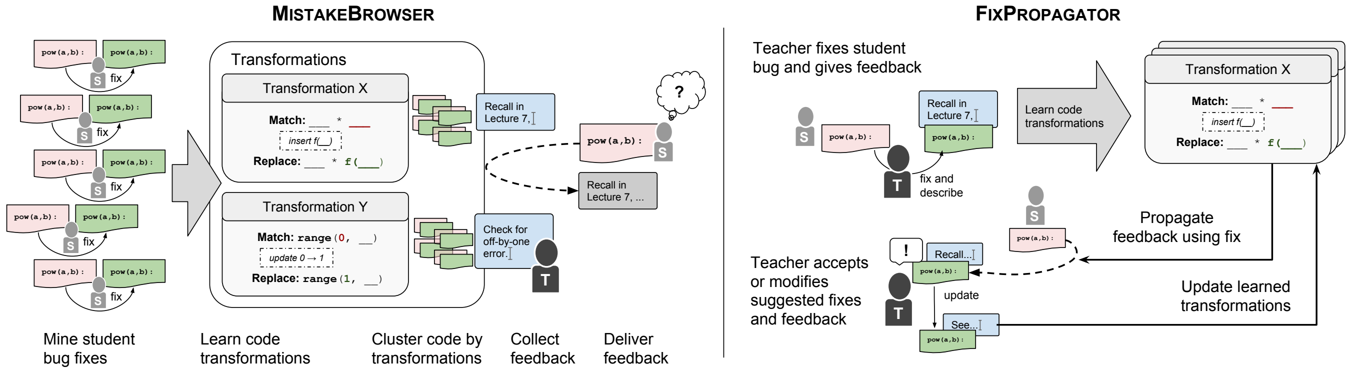


Figure 1. We contribute two interfaces that help teachers give feedback on incorrect student submissions using program synthesis. MISTAKEBROWSER learns code transformations from examples of students fixing bugs in their own code. Using these transformations, MISTAKEBROWSER clusters and fixes current and future incorrect submissions. The teacher adds feedback, one cluster at a time. FIXPROPAGATOR learns code transformations from examples of teachers fixing bugs in incorrect student submissions. The teacher annotates each fix with feedback. Using these annotated transformations, FIXPROPAGATOR propagates fixes and feedback to current and future incorrect submissions.

ABSTRACT

In large introductory programming classes, teacher feedback on individual incorrect student submissions is often infeasible. Program synthesis techniques are capable of fixing student bugs and generating hints automatically, but they lack the deep domain knowledge of a teacher and can generate functionally correct but stylistically poor fixes. We introduce a mixed-initiative approach which combines teacher expertise with data-driven program synthesis techniques.

We demonstrate our novel approach in two systems that use different interaction mechanisms. Our systems use program synthesis to learn bug-fixing code transformations and then cluster incorrect submissions by the transformations that correct them. The MISTAKEBROWSER system learns transformations from examples of students fixing bugs in their own submissions. The FIXPROPAGATOR system learns transformations from teachers fixing bugs in incorrect student submissions. Teachers can write feedback about a single submission or a cluster of submissions and propagate the feedback to all other submissions that can be fixed by the same transformation. Two studies suggest this approach helps teachers better understand student bugs and write reusable feedback that scales to a massive introductory programming classroom.

Author Keywords

programming education; program synthesis

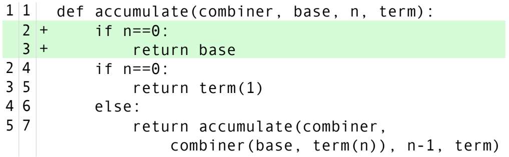
INTRODUCTION

One of the most common forms of instantaneous debugging assistance in Computer Science (CS) classes is the autograder. Typical autograders display the results of running a teacher-written test suite on student *submissions*, or solution attempts to programming exercises. However, feedback in the form of test results leaves students with a large gulf of evaluation [16] between the failed tests and bugs in their own submissions.

Recently developed techniques address this gulf by automatically generating or learning *bug fixes*, or code edits, for a submission such that a modified submission passes all tests [11, 19, 22, 20, 13]. From each fix, a personalized hint can be generated. The hints range from vague pointers, e.g., identifying the faulty line, to specific instructions about what new expression should replace the faulty one. Most of these approaches use program synthesis [7] to generate the fixes and hints.

L@S 2017, April 20-21, 2017, Cambridge, MA, USA
ACM 978-1-4503-4450-0/17/04.
http://dx.doi.org/10.1145/3051457.3051467

∗. These three authors contributed equally to the work.



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

Figure 2. A functionally correct but stylistically poor bug fix generated by program synthesis.

These automated techniques suffer from two key flaws. First, the hints lack the deep domain knowledge of a teacher. They do not address the underlying misconceptions of students or point to relevant principles or course materials. Second, the fixes can be functionally correct but stylistically poor and therefore potentially misleading when used as the basis of a hint. For example, Fig. 2 shows a bug fix synthesized with Refazer [20]. The fix inserts a nearly identical correct base case immediately before the incorrect base case. This is poor coding practice, and hints generated from this fix may be misleading. Specifically, hinting at this fix by its location, such as *Add code before line 2*, would be misleading because the bug is in the return statement on line 3. This is not a one-time occurrence: in our study, teachers rejected, on average, 19% ($\sigma = 18\%$) of the fixes synthesized by Refazer.

We introduce a mixed-initiative approach which allows teachers to combine their deep domain knowledge with the results of data-driven program synthesis techniques. The teacher and program synthesis back-end take turns applying their relative strengths in pursuit of the common goal of providing reusable, teacher-written feedback at scale. We demonstrate and evaluate this approach in two novel systems, MistakeBrowser and FixPropagator, which illustrate two different interaction mechanisms.

MistakeBrowser and FixPropagator both rely on Refazer, a data-driven program synthesis technique that learns *code transformations* from examples of bug fixes. Transformations are sequences of rewrite rules applied to the abstract syntax tree (AST) of a program. In our systems, we use transformations to: (1) cluster incorrect submissions, so cluster members may share a common bug or misconception; (2) generate bug fixes for each incorrect submission; (3) propagate teacher-written feedback to all incorrect submissions that are fixed by the same transformation. These learned code transformations can be reused to apply feedback to students through autograders or apply rubric items to exam submissions in current and future semesters.

In the MistakeBrowser system, code transformations are learned from examples of student-written bug fixes. The history of student attempts is already available today in many autograding systems. In the MistakeBrowser workflow, shown on the left in Figure 1, the teacher reviews the incorrect student submissions that were clustered offline by the code transformations that corrected them. Each submission is shown as a diff between its incorrect and corrected form. Even if some synthesized bug fixes are stylistically poor, the incorrect submissions in any one cluster may all share a misconception because the same code transformation corrected all of them. When reviewing the cluster, the teacher can infer the shared misconception and write feedback for the whole cluster that includes explanations, hints, or references to relevant course materials.

When submission histories are not available, FixPropagator can learn code transformations from teachers fixing bugs in incorrect student submissions. As shown on the right in Figure 1, the process is cyclic and iterative. First, the teacher demonstrates a stylistically good fix that corrects a single incorrect submission. Then they annotate the fix with feedback. For each bug fix and annotation the teacher enters, a synthesis back-end learns and propagates these fixes and feedback to more incorrect submissions in the dataset. The teacher reviews these propagated fixes and feedback as pending suggestions and accepts them or modifies them as necessary. Modifications kick off another round of synthesis to learn and propagate the updated fix and feedback to more incorrect submissions.

Since MistakeBrowser requires prior student data and the FixPropagator workflow does not, the FixPropagator system is most suitable for newly introduced homework or exam programming problems. While it is not currently standard practice for teachers to explicitly fix bugs in incorrect submissions, doing so with FixPropagator allows teachers to scale their ability to provide manually written feedback — and grades — by propagating already authored feedback to current and future incorrect submissions.

We ran two user studies, one for each system. Seventeen current and former teachers from the staff of a massive introductory programming class participated. In both studies, the teachers were using MistakeBrowser or FixPropagator to interact with incorrect student submissions collected from previous semesters of the same class. The study of MistakeBrowser suggests that the system helps teachers understand what mistakes and algorithms are common in student submissions. Teachers appreciated the generated fixes, but confirmed that a human in the loop is needed to review and annotate them with conceptual or high-level feedback. In the study of FixPropagator, teachers wrote fixes and feedback for dozens of incorrect submissions, which the system propagated to hundreds of other submissions. Teachers generally accepted propagated fixes and feedback, although some feedback needed to be rewritten to generalize to additional submissions. Together, the studies suggest that our approach helps teachers better understand student bugs and provide scalable and reusable feedback.

This paper makes the following contributions:

- A technique for clustering incorrect submissions by the code transformations that correct them.
- A mixed-initiative approach in which teachers combine domain knowledge with the results of data-driven program synthesis.
- Two systems that demonstrate this approach with different interaction mechanisms.
- Two studies that suggest this approach helps teachers better understand student bugs and write reusable feedback that scales to a massive introductory programming classroom.

RELATED WORK

Feedback is critical to the learning experience [1], and teachers can be excellent sources of personalized, timely feedback [12]. As the class size grows, personal attention becomes infeasible; however, students in larger groups may share common errors and misconceptions [5]. Relevant prior work falls into two general categories: systems that rely on instructors to generate feedback, but provide better tools to do so; and systems that compute and display feedback automatically.

Tools that Support Instructor Feedback at Scale. Several user interfaces and systems empower teachers to manage large numbers of programming students. CodeOpticon [8] enables instructors to monitor many students simultaneously and provide situated help on code-in-progress. OverCode [6] and Foobaz [4] normalize and cluster correct student submissions so teachers do not need to read thousands of submissions to identify common and uncommon student choices about syntax and style. AutoStyle [14] clusters correct submissions using a metric of code complexity so that teachers can write hints how the code in a cluster can be written more simply. Singh et al. [21] define a problem-independent grammar of features; a supervised learning algorithm trained on teacher-graded examples can map new student code submissions to grades. However, these systems cannot give a teacher a high-level overview of the different, common misconceptions and bugs their own students have, like MISTAKEBROWSER can.

Clustering Student Submissions and Bugs. Identifying and clustering semantically similar student code submissions in a robust, general way is a challenge. Nguyen et al. [15] define probabilistic semantic equivalence to cluster functionally equivalent but syntactically distinct code phrases within submissions. Piech et al. [17] use neural networks to learn program embeddings and cluster submissions. Kaleeswaran et al. [11] cluster dynamic programming (DP) submissions by ‘solution strategy,’ using static analysis to detect how students manipulate arrays that store the results of subproblems in a DP solution. Earlier work relies on clustering student submissions using various distance metrics, like AST tree edit distance [10]. Instead of clustering code, Glassman et al.’s “learnersourcing” workflow [5] and HelpMeOut [9] cluster bug fixes by failed test cases, compiler errors, or runtime exceptions. Rather than clustering based on behaviorial or syntactic similiarity, our approach is to cluster incorrect submissions based on the transformation that corrects them.

Algorithmically Generating Debugging Feedback. Intelligent Tutoring Systems (ITS) seek to emulate one-on-one tutoring and provide personalized feedback by using rule-based or constraint-based methods [23]. However, traditional rule-based feedback requires much time and expert knowledge to construct [19]; it does not scale well for programming exercises, which have large and complex solution spaces.

Data-driven methods have recently been introduced to augment existing techniques. Rivers et al. [19] use student data to incrementally improve ITS feedback for Python assignments. Codewebs [15] and Codex [3] use machine learning to analyze large volumes of code, extract patterns, flag anomalies as possible errors, and, if deployed in an educational context, could deliver feedback. These techniques can leverage the statistical properties of large numbers of student submissions, but they suffer from the cold-start problem. FIXPROPAGATOR enables teachers to provide examples to bootstrap hint generation.

Program Synthesis for Feedback Generation. Recent advances in program synthesis can help programming teachers and students in verifiably correct ways that statistical or rule-based techniques cannot. AutomataTutor [2] uses program synthesis to generate conceptual hints in the domain of automata constructions. Synthesized bug fixes have also been used to generate personalized hints for introductory-level programming assignments [11, 22]. AutoGrader [22] can find a minimal sequence of “repairs” that transforms a student’s incorrect solution into a correct one; however, it requires that the teacher manually write down an error model of possible local modifications ahead of time. Instead of requiring a hard-coded error model, Lazar et al. [13] mine textual line edits from student interactions with a Prolog tutor, and synthesize code fixes by combining these edits. Rolim et al. [20] take an example-based approach to learn code fixes as abstract syntax tree transformations from pairs of incorrect and correct student submissions. We build on this technique in this paper.

WRITING REUSABLE CODE FEEDBACK WITH MIXED-INITIATIVE INTERFACES

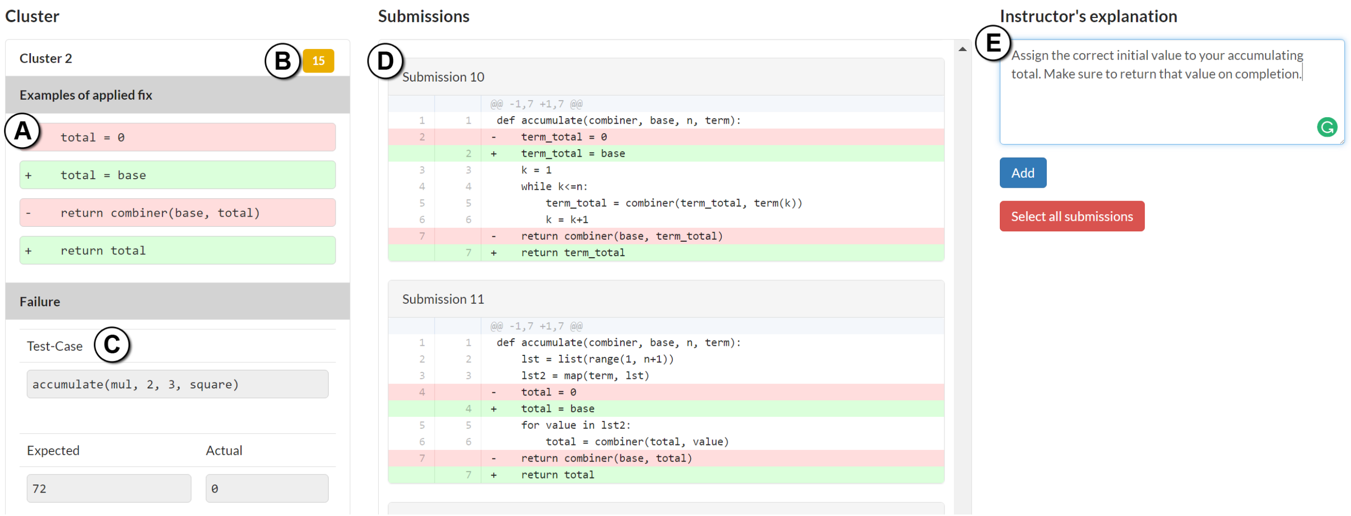
We established the following design goals based on our literature review and our understanding of current pain points in large programming courses: **(1)** Help teachers better understand the distribution of common student bugs in introductory programming assignments. **(2)** Help teachers understand the nature of those bugs and ways to fix them. **(3)** Help teachers scale teacher-authored feedback to large numbers of students in a way that is reusable across semesters. We first briefly review how program synthesis enables MISTAKEBROWSER and FIXPROPAGATOR, then describe both systems.

Using Program Synthesis To Cluster Submissions

To reduce teacher burden, our systems automatically find groups of student submissions that exhibit the same underlying problem. We extract code transformations from pairs of incorrect and correct student submissions. We then check if a transformation can be successfully applied to other incorrect student submissions. Success is defined relative to an assignment’s test suite: a transformation is successful if applying the transformation makes the corrected solution pass all tests.

In MISTAKEBROWSER, the pairs of incorrect and correct code come from histories of student submissions to an autograder that culminate in a correct submission. In FIXPROPAGATOR, the pairs of incorrect and correct code come from the small subset of incorrect student submissions that teachers choose to manually correct (see Figure 1).

Naïve extraction of code transformations through simple text differencing or abstract syntax tree differencing does not work well. Consider the two incorrect submissions in Figure 3, center column: while they are conceptually similar, and indeed exhibit the same underlying problem, they differ both in variable names and in code structure — one uses a loop with an index variable, and the other uses list iteration. Thus, it is important



Cluster 2 15

Examples of applied fix

```
-     total = 0
+     total = base
-     return combiner(base, total)
+     return total
```

Failure

Test-Case

`accumulate(mul, 2, 3, square)`

Expected: 72

Actual: 0

Submission 10

```
@@ -1,7 +1,7 @@
 def accumulate(combiner, base, n, term):
-    term_total = 0
+    term_total = base
     k = 1
     while k<=n:
         term_total = combiner(term_total, term(k))
         k = k+1
-    return combiner(base, term_total)
+    return term_total
```

Submission 11

```
@@ -1,7 +1,7 @@
 def accumulate(combiner, base, n, term):
     lst = list(range(1, n+1))
     lst2 = map(term, lst)
-    total = 0
+    total = base
     for value in lst2:
         total = combiner(total, value)
-    return combiner(base, total)
+    return total
```

Assign the correct initial value to your accumulating total. Make sure to return that value on completion.

Figure 3. MISTAKEBROWSER interface: On the left panel, teachers can find information about the current cluster, such as an example of the synthesized fix (A); the total number of submissions in the cluster (B); the failing test case input, the expected output, and the actual output produced by the incorrect submissions (C). The center column shows the incorrect submissions before and after the synthesized fix (D). Finally, on the right panel, instructors can add explanations about student mistakes (E).

to find *abstract* transformations that capture edits at a level that can be reused across different students. In our example, the abstract transformation expresses that a student replaced the `0` on the right-hand-side of an assignment with function parameter `base`; and the function call inside the `return` statement should be replaced with the second argument inside that call (e.g., replace `return combiner(base, total)` to `total`).

We generate abstract code transformations using Refazer [20], which in turn builds on the PROSE synthesis framework [18]. Refazer uses a Domain-Specific Language (DSL) to specify transformations, and synthesizes transformations as programs in that DSL that map from incorrect to correct submissions. The language allows abstracting nodes in the Abstract Syntax Tree (AST) of a submission using a tree pattern matching language. It then offers common tree edit operations to modify nodes in the AST, such as *Insert*, *Delete*, *Update*, and *Move*. Returning to our example, the transformation synthesized by Refazer to fix submissions 10 and 11 in Figure 3 has 3 AST operations: (i) *Update* a constant value to `base`; (ii) *Delete* a function call with two name arguments located in a return statement and (iii) *Move* the second argument of this call to the beginning of the return statement.

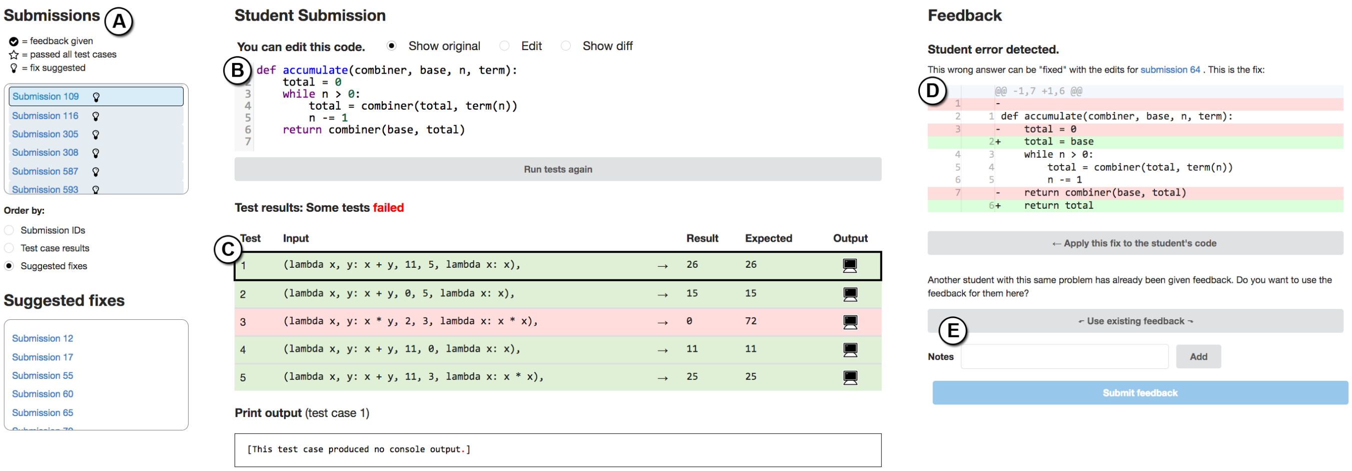
Browsing Student Bugs

Consider the teaching staff of a massive introductory programming class, CS1, that have been using the same programming assignments for weekly 'finger exercises' every semester for years. These exercises are intended to reinforce new concepts introduced in class each week. Since teaching staff are not present when students attempt these exercises, they do not know what bugs and misconceptions are most common in student code, except through student forum posts.

Before the semester starts, Jamie, the lead teaching assistant (TA), loads student code snapshots from the prior semester from the class autograder into MISTAKEBROWSER, shown in Figure 3. The back-end learns reusable abstract transformations from the bug fixes made by students in previous semesters. The MISTAKEBROWSER interface displays, one at a time, clusters of incorrect submissions that are corrected by the same transformation, along with their synthesized fixes. The center pane lists all incorrect submissions in that cluster, showing incorrect code fragments in red, and fixes in green in a common code difference view (Figure 3D). Jamie reviews each cluster and writes down conceptual feedback for each cluster. To explain the cluster's contents, the interface shows a compact representation of the fix for the cluster (Figure 3A), how many incorrect submissions comprise the cluster (Figure 3B), and the return value or exception of a representative submission for the first test case it fails to pass (Figure 3C). There are two clustering variants we consider in the user study that follows. Figure 3 shows the CLUSTERBYFIXANDTESTCASE variant, in which incorrect submissions are clustered both by the transformation that fix them and by the return value of the first failed test case. In the CLUSTERBYFIX variant, incorrect submissions are clustered only by transformation.

After reviewing a cluster, Jamie composes high-level feedback in free-form text that applies to all submissions in the cluster (Figure 3E). For the cluster shown in Figure 3, she might write the hint, *Assign the correct initial value to your accumulating total, and make sure you return that value on completion.* When Jamie is satisfied that the most common and interesting clusters have been annotated with explanations, hints, or references to relevant course materials, MISTAKEBROWSER can be left running as part of the course autograder's back-end, where it can deliver the TAs' feedback to students during current and future semesters, along with the test case successes and failures, whenever an incorrect submission falls into an annotated cluster in MISTAKEBROWSER.

MISTAKEBROWSER clusters are based on program transformations synthesized by Refazer. For each homework assignment, the back-end keeps a list of Refazer transformations and the



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

Test results: Some tests failed

Test	Input		Result	Expected	Output
1	(lambda x, y: x + y, 11, 5, lambda x: x),	→	26	26	🖥
2	(lambda x, y: x + y, 0, 5, lambda x: x),	→	15	15	🖥
3	(lambda x, y: x * y, 2, 3, lambda x: x * x),	→	0	72	🖥
4	(lambda x, y: x + y, 11, 0, lambda x: x),	→	11	11	🖥
5	(lambda x, y: x + y, 11, 3, lambda x: x * x),	→	25	25	🖥

Print output (test case 1)

```
[This test case produced no console output.]
```

Student error detected.

This wrong answer can be "fixed" with the edits for submission 64 . This is the fix:

```
@@ -1,7 +1,6 @@
-
 def accumulate(combiner, base, n, term):
-    total = 0
+    total = base
     while n > 0:
         total = combiner(total, term(n))
         n -= 1
-    return combiner(base, total)
+    return total
```

Another student with this same problem has already been given feedback. Do you want to use the feedback for them here?

Figure 4. FIXPROPAGATOR interface: The left panel shows all of the incorrect submissions (A). When the teacher selects one, the submission is loaded into the Python code editor in the center of the interface (B). Then the teacher can edit the code, re-run tests, and inspect results. The bottom of the center panel shows the list of tests and console output (C). Once the teacher has fixed the submission, they add some hint that will be shown to current and future students fixed by the same transformation. The bottom of the left panel shows submissions for which the system is suggesting a fix. When the teacher selects a suggested fix, it is shown as a diff in the right panel (D). The teacher can reuse the previously written hint or create a new one (E).

assignment's test suite. Given an incorrect submission, the system iterates over the list of transformations, and for each transformation, tries to apply it and checks whether the code is fixed according to the test suite. As soon as the system finds a transformation that fixes the submission, it adds the submission to the cluster associated with this transformation. In the CLUSTERBYFIXANDTESTCASE, the system uses additional information provided by the tests related to the actual and expected outputs to create clusters.

Propagating a Teacher's Bug Fixes

Sam, the lead TA of a massive introductory programming class at another school, wants to deploy the same kind of high-level feedback on incorrect submissions that MISTAKEBROWSER enables. However, their course infrastructure only saves the most recent submission from each student, so there is no history of student bug fixes from which MISTAKEBROWSER could learn transformations. Instead, Sam uploads the incorrect submissions he has to FIXPROPAGATOR. Figure 4 shows the FIXPROPAGATOR user interface.

In the FIXPROPAGATOR interface, Sam looks at incorrect submissions by selecting them (Figure 4A), iteratively edits and executes the submission in an interactive code editor (Figure 4B) against the teacher's test suite (Figure 4C), and adds some high-level feedback for the student, such as explanations, hints, or pointers to relevant course materials. Ideally, this feedback should be worded so that future students in need of a similar fix would also find it beneficial (Figure 4E).

When Sam submits feedback, the system uploads the original incorrect submission, fixed submission, and high-level feedback to a synthesis back-end to learn generalized transformations from Sam's correction. FIXPROPAGATOR applies each transformation to the incorrect submissions that do not yet have feedback. Transformations that fix incorrect submissions turn into suggested fixes–along with the corresponding feedback–in the FIXPROPAGATOR interface (bottom of Figure 4A) that can be accepted with a single click (Figure 4D). If accepted, the tests are run automatically and Sam sees that, indeed, this fix is just what the student needs to correct their submission. Sam clicks on a button to reuse the feedback from the submission that generated the fix (Figure 4E). If Sam judges the fix or the feedback as not appropriate, it can be modified in place. Changes to synthesized fixes become new bug fix examples that spur the generation of new transformations in the back-end. Sam alternates between reviewing suggestions and manually correcting more incorrect submissions. After a while, most submissions have suggestions.

Given the high cost of debugging student code, a teacher should be able to fix few incorrect submissions and see feedback propagate to many other students. We modified Refazer to synthesize generalizable fixes from just one fix. To improve generalization, Refazer produces multiple transformations of varying generality for each submitted fix. All generated rules are applied to all submissions that have not yet been fixed.

Furthermore, FIXPROPAGATOR needs to support online fix generation at interactive speeds. However, effectively searching a space of code transformations can be time-consuming; with the current synthesis back-end, it can take minutes to synthesize and apply fixes to other submissions. The user interface was decoupled from Refazer so that teachers can continue to fix and test code, produce feedback, and move on to other submissions while the back-end discovers fixes. All communication with the synthesis back-end aside from initialization is asynchronous: fixes are uploaded to and retrieved from Refazer using background threads.

Our implementation anticipates future modifications to support collaborative production of fixes and feedback. Communications with Refazer are moderated as "sessions" sharing a common set of submissions and synthesized transformations. We have made the code for both the web server and front end available under an open source license[1].

[1] https://github.com/ace-lab/refazer4CSteachers

USER STUDIES

We ran two in-lab user studies with the teaching staff of a massive programming class, with one study per system. The studies evaluate how effective the interfaces are at helping teachers understand common bugs and write feedback to help students overcome them.

Participants

We recruited 17 teachers from the pool of current and former CS61a teaching staff members. CS61a is a massive introductory programming class at UC Berkeley with as many as 1,500 students enrolled per semester. All the teachers in our study are over 18 years old (average: 19.76 years old, σ =1.39). 16 of the 17 teachers are currently serving on the class teaching staff, and the remaining participant was previously a class teaching assistant for many semesters.

We split the pool of teachers into two groups of size 9 and 8. The first group (SS1-9) tested MISTAKEBROWSER, while the second group (ST1-8) tested FIXPROPAGATOR. The identifiers SS and ST reference the fact that bug fixes in MISTAKEBROWSER and FIXPROPAGATOR are student and teacher-generated, respectively. All teachers were qualified to try both tools, but we limited each teacher to one system due to the length of time required to thoroughly evaluate a system.

Dataset

Whenever a CS61a student submits code to be tested by the course autograder against the teacher-written test suite, the system logs the code, student ID, and test results. From one homework assigned in Spring 2015, we selected the three programming exercises below.

Product (data from 549 students): takes as parameters a positive integer n and a unary function $term$, and returns the product of the first n terms in a sequence: $term(1) * term(2) * ... * term(n)$.

Accumulate (668 students): takes as parameters the same n and $term$ as Product as well as a binary function $combiner$ for accumulating terms, and an initial value $base$. For example, $accumulate(add, 11, 3, square)$ returns $11 + square(1) + square(2) + square(3)$.

Repeated (720 students): takes as parameters a unary function f and a number n, and returns the nth application of f. For example, $repeated(square, 2)(5)$ returns $square(square(5))$, which evaluates to 625.

For each exercise, the interfaces were populated with incorrect submissions using a two-step process. First, we extracted each student's final correct solution and preceding incorrect submission. Using these submission pairs as examples, we trained Refazer to synthesize code transformations to fix common bugs. Second, for each student, we identified the earliest incorrect submission that the transformations could fix. These early, fixed submissions were shown to teachers in clusters of fixed submissions in MISTAKEBROWSER, or as incorrect submissions in FIXPROPAGATOR.

When pre-populating MISTAKEBROWSER, Refazer generated mostly small synthesized fixes for the dataset: on average, the tree edit distance between the abstract syntax trees of incorrect and fixed submissions was 4.9 ($\sigma = 5.1$).

Shared Protocol: Setup and Training

Teachers were invited to an on-campus lab for one hour and offered 20 US dollars in exchange for their time and expertise. The experimenter walked the teacher through the features of the interface they would see, demonstrating actions on one of the incorrect submissions that the teacher would be working on. This walk-through included a few minutes of explanation about the synthesis back-end. We chose to give this brief explanation because, during pilot studies, teachers who did not receive an explanation were distracted from the task by their own curiosity and theories about the back-end's inner workings. The tutorial took no more than five minutes. Finally, the experimenter walked the teacher through a brief description of the first programming exercise for which the teacher will see incorrect submissions: the purpose of the programming exercise, the test cases used to check the correctness of submissions, and the expected test case return values.

STUDY 1: MISTAKEBROWSER

The purpose of this user study was to evaluate the MISTAKEBROWSER system. We asked the following research questions: **(1)** How do teachers perceive the quality of synthesized fixes? **(2)** Do synthesis-based clusters help teachers write feedback? **(3)** How reusable is the cluster-based feedback?

Study Protocol

Teachers had 40 minutes to review clusters of incorrect submissions and write feedback for each cluster. They viewed two clustering interface variants, CLUSTERBYFIX and CLUSTERBYFIXANDTESTCASE, for 20 minutes each. The order of interface variants and choice of programming exercise from the three exercises were counterbalanced across teachers. Teachers were assigned an interface variant, problem, and cluster to start with. For each cluster, they marked all of the poor synthesized fixes to student submissions. They then answered a few questions about the semantic coherence of the cluster, e.g., "Do these incorrect submissions share the same misconception?" They were asked to "write the most precise short description [they] can of the fix [they] would suggest," which need not match the synthesized fix. The teacher also answered Likert scale questions about their confidence in their descriptions and the depth of domain knowledge they added in the process. As soon as they finished these tasks for a cluster, they could advance to the next of the largest three clusters in their assigned programming exercise and interface. After the second 20-minute period, teachers reflected on their experiences in a final survey.

Results

Refazer, our synthesis back-end, generated fixes for 87% of the students in our dataset, resulting in an average of 549 fixes across all three problems. On average, these fixes were grouped into 134 clusters in the CLUSTERBYFIX condition and 198 clusters in the CLUSTERBYFIXANDTESTCASE condition. Within the top three clusters for all programming exercises, the largest cluster contained, on average, 109 submissions and the smallest cluster contained 32 submissions.

Teachers saw an average of 3 ($\sigma = 1.4$) clusters containing 145 ($\sigma = 80.9$) incorrect submissions per hour-long session, where they spent 20 min in each clustering condition. They saw, on average, 72.9 ($\sigma = 53.4$) incorrect submissions in the ClusterByFix condition and 78.7 ($\sigma = 43.9$) incorrect submissions in the ClusterByFixAndTestCase condition.

Perceived quality of synthesized fixes. On average, 19% ($\sigma = 18$) of the synthesized fixes were considered poor by the teachers, and all teachers reported at least one poor fix. This corroborates our initial intuition that synthesis alone is not enough to generate high-quality hints.

For instance, SS2 and SS7 noticed a synthesized fix that did not match the approach teachers explicitly taught for coding recursive solutions. SS2 called the suggested fix "dangerous." SS2 also called some fixes "hot fixes" because, after application, the student submission returned the expected values but still had logical failures. In one example, the system suggested a fix that compensated for, rather than corrected, an incorrect variable initialization. Some fixes were not full fixes, even though the fixed code passed all the teacher-written test cases. For example, some incorrect submissions computed the product of $term(i)$ for $i = 1..n-1$, instead of $1..n$. A synthesized "hot fix" changed the range of the loop to $i = 2..n$. However, this incorrect logic was not caught by the teacher's test suite: $term(1)$ returned 1 for all $term$ functions in the test cases, so it was impossible to detect if the student failed to call $term$ when $i = 1$. This result revealed limitations in the teacher-written test suite. SS8 noticed this and marked the entire cluster of fixes as poor.

The value of clustering by code transformation. While not all synthesized fixes were appropriate for students, teachers did appreciate seeing them. When completing free response questions about what they liked in the interfaces, two-thirds of teachers (six of nine) specifically named clustering by transformation as a feature they appreciated and two-thirds named the synthesized fixes directly. All nine teachers named at least one of those two features as what they appreciated most.

Teachers described how the synthesized fixes, shown as highlighted diffs, helped during their task: "highlight[ing] the part of the code that was incorrect ... made it much easier to quickly learn what was wrong with the code and how to fix it" (SS7). These diffs were "fast and easy to review" and "familiar" (SS3).

Subject SS1 wrote, "I thought it was interesting how grouping student answers by their common mistakes actually revealed something about the misconceptions they shared!" The utility of this clustering was apparent to SS3: "Seeing all of the similar instances of the same (or nearly the same) misconception was very useful, because it suggested ways to address common issues shared by many students." They agreed with the statement "These interfaces gave me insight into student mistakes and misconceptions" at the level of 6.2 ($\sigma = 0.44$) on a scale from 1 (strongly disagree) to 7 (strongly agree); no teacher rated their agreement as lower than a 6.

Several teachers' responses support the hypothesis that MistakeBrowser gives a high-level view of the misconceptions and bugs students labor under while solving the problem, like an OverCode [6] for incorrect submissions. SS9 liked that "it had a wide variety of student responses to the same problem." SS1 wrote, "I felt that being able to compare many different solutions (i.e. iterative, recursive, tail-recursive) was insightful as to how the students approached the problem."

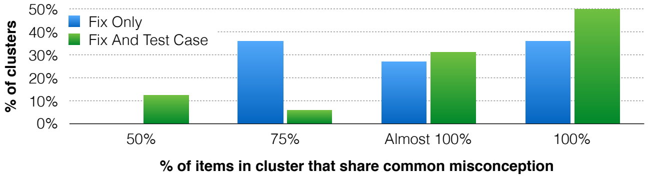


Figure 5. Distribution of clusters with respect to % of solutions that shared common misconceptions.

Reusability of the feedback. To evaluate the reusability of feedback assigned to synthesis-based clusters, we asked teachers to report how many submissions in each cluster actually shared the same misconception. Figure 5 shows teachers' answers for the two cluster conditions, ClusterByFix and ClusterByFixAndTestCase. In both conditions, they reported that most submissions share the same misconceptions. However, they reported a greater proportion of ClusterByFixAndTestCase clusters as "100%" or "100% with a few exceptions" compared to ClusterByFix clusters. Seven out of nine teachers also mentioned in the final survey they preferred ClusterByFixAndTestCase cluster because the combination of fixes and test cases made it easier to check if the incorrect submissions share the same misconception.

One of the clusters for the ClusterByFixAndTestCase condition was reported to be less internally consistent than the others, with only "50%" of submissions sharing a common misconception. As we learned from teachers in the study, the provided test cases failed to reveal a bug that caused a subset of submissions to behave differently than the other submissions. Submissions with this bug required a different fix.

STUDY 2: FixPropagator

The purpose of this user study was to evaluate FixPropagator. Specifically, we had the following research questions:
(1) Can FixPropagator propagate a small number of teacher-written fixes and feedback to many incorrect submissions?
(2) Can FixPropagator's back-end perform fast enough to support real-time interaction?
(3) Do teachers accept propagated fixes and hints?

Study protocol

Each teacher was assigned to review student submissions for one of the three programming exercises. After a five-minute tutorial with the FixPropagator interface, they were given thirty minutes to interact with the system to teach the system to fix and provide feedback on student code. This thirty-minute period was broken up into alternating five-minute tasks. In the first five-minute task, the teacher was asked to fix as many bugs as possible, to maximize the number of generated fixes. The experimenter told the teacher that simpler bug fixes may yield more suggested fixes. During the second five-minutes task, the

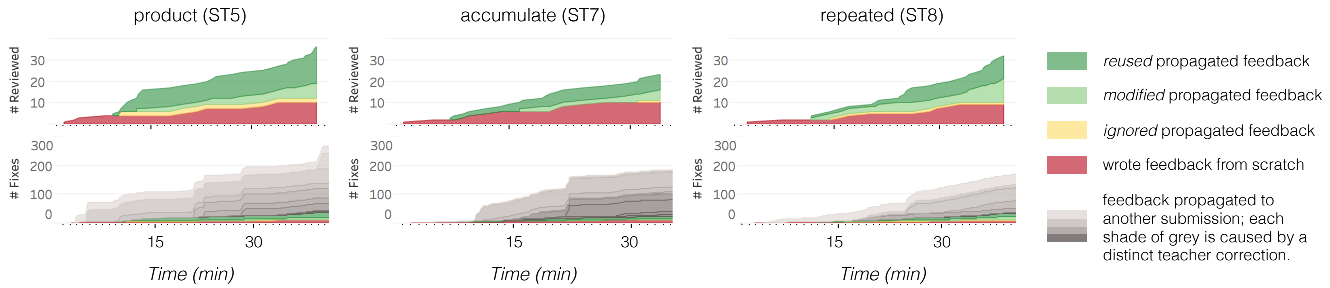


Figure 6. Top row: The number of incorrect submissions for which teachers provided feedback, shown for three teachers (ST5, ST7, ST8). Bottom row: The number of incorrect submissions that received propagated feedback, using a teacher's hand-written fixes and feedback (ST5, ST7, ST8).

teacher reviewed pending fixes and then accepted or modified them, so that they could check whether the system had learned acceptable transformations for fixing incorrect submissions they had not yet seen. After the thirty-minute period, teachers were asked to fill out a post-study reflection survey, including Likert-scale and free response questions about their experience with FixPropagator.

Results

During this study, due to unforeseen circumstances, our synthesis back-end was disabled for part of two teachers' sessions (ST1, ST4). Data described here was collected for the remaining six teachers.

Bug fix propagation. Teachers provided examples of bug fixes in two ways: fixing incorrect submissions from scratch, and editing suggested fixes. They fixed a median of 10 ($\sigma = 2.7$) submissions from scratch and fixed 3 ($\sigma = 2.9$) more after editing suggested fixes. During the time of the study session and up to 40 minutes after the study, our synthesis back-end was able to fix a median of 201 submissions ($\sigma = 47.7$). Figure 6 shows the propagation of fixes over time. By the end of the study, a large portion of the submissions (average = 34.7%, $\sigma = 10.19\%$) had either been corrected by the teacher or fixed by a synthesized transformation.

Performance. It took a median of 2 minutes and 20 seconds ($\sigma = 7m34s$) to successfully propagate a fix to another submission after a teacher corrected an incorrect submission. Although the current system does not immediately show teachers suggestions based on their corrections, teachers were able to work on other submissions while waiting for synthesized fixes. Figure 7 shows the interaction of one of the teachers with FixPropagator. Teachers alternated between fixing and reviewing, but the effort invested in manual fixes allowed them to accept a large number of auto-propagated fixes.

The value of synthesized fixes. Transformations learned from teachers' manual corrections fixed new incorrect submissions in unexpected ways. They helped teachers better understand the space of bug fixes and approaches to implementing the solution. For example, ST3 came across an incorrect submission which was very close to being correct. She did not see the simple fix and instead wrote an elaborate fix that was fundamentally different from the student's approach. Later, a simpler synthesized fix to a similar incorrect submission was suggested, and she realized the submissions were using a different but valid approach to solving the problem. After accepting the suggested fix, she reported she had learned something about the space of solutions for the exercise.

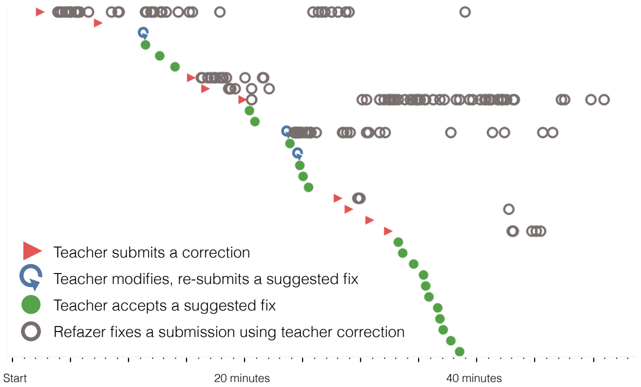


Figure 7. Timeline of the corrections a teacher (ST8) made to incorrect submissions, and the subsequent synthesized fixes that were generated from each correction.

Reusability. After contributing fixes and feedback for incorrect student submissions, teachers generally accepted the fixes and feedback propagated to other submissions. Teachers were more likely to reuse propagated fixes to the code verbatim (median = 17 times, $\sigma = 8.9$) than to reuse feedback verbatim (median = 11, $\sigma = 6.3$). Fixes were likely propagated correctly more often than feedback, as fixes were only propagated if they allowed a submission to pass test cases that it failed before. However, teachers' feedback did not always generalize to new submissions. Some feedback referred to arbitrary implementation choices not present in other submissions. For example, one teacher referenced a specific variable name when writing, "Your starting value of z should be a function, not an int." When proposed fixes and feedback were not enough, teachers made modifications after applying suggested fixes (median = 3, $\sigma = 2.9$), and modified the feedback (median = 6, $\sigma = 2.7$).

Survey responses confirmed our observations about the acceptability of fixes and feedback. Most teachers reported that pending fixes were acceptable "100% of the time, with a few exceptions." The proportion of acceptable feedback was one category worse: teachers rated the suggested feedback as accurate "75%" of the time.

DISCUSSION

Our first design goal is to help teachers better understand the distribution of common student bugs in introductory programming exercises. MISTAKEBROWSER achieved this goal by clustering incorrect submissions by the transformation that corrects them. Given that the teachers in our studies had no comparable view of incorrect submissions, this added significant value. FIXPROPAGATOR achieved this goal indirectly, by helping teachers discover how many different incorrect submissions could be fixed with the same transformation.

Our second design goal is to help teachers understand the nature of student bugs and ways to fix them. Both MISTAKEBROWSER and FIXPROPAGATOR achieve this goal by visualizing the synthesized fix as a diff for every incorrect submission in each cluster. By seeing the variety of submissions fixed by a common transformation, teachers begin to understand the essence of the underlying misconception, as well as the variety of submissions it can appear in.

Our third design goal is to give teachers a tool for composing high-level feedback and hints that scale to large numbers of students and can be reused in future semesters. Teachers can achieve this goal with either system, depending on the availability and quality of archives of student debugging activity. As seen in the first study, the existence of poor synthesized fixes within a cluster does not prevent teachers from composing high-level feedback that can be propagated to current and future incorrect submissions in that cluster. Clustering by transformation and test cases reduced cluster size but increased cluster purity, as in, there was more likely to be a single bug shared across all incorrect submissions in the cluster. Despite the smaller size of clusters in the CLUSTERBYFIXANDTESTCASE variant of MISTAKEBROWSER, teachers still reviewed as many or more incorrect submissions. In the FIXPROPAGATOR system, after only a few minutes of manually fixing and providing feedback on a few incorrect submissions, teachers received bug fix and feedback suggestions for tens or hundreds of additional incorrect submissions. Even if only a large minority of students receive high-level feedback in which teachers can remind the student of relevant principles and course content, it is still a major advance over the status quo of feedback for students in massive programming classes.

One participant (SS3) mentioned that they used to hand-grade homework submissions, giving feedback as well as grades, until their class became too large. Now they only evaluate student homework based on a proxy for student effort, test cases passed, and spot-checks for composition. He thought that MISTAKEBROWSER could help the staff grade their massive class the same way they used to grade homework when the class was smaller. The FIXPROPAGATOR system can also be used for grading-through-debugging. Debugging a student exam submission is not a trivial activity, but FIXPROPAGATOR can potentially learn reusable transformations from every successful correction, simplifying adjustments to the grading rubric and point deductions during the grading process. If the exam problem is reused, the problem-specific rubric and point deduction for that problem can be reused and added to during future exam grading sessions, bringing the staff closer to fully automatic submission grading.

Limitations. Our studies do not evaluate the impact of these systems on student learning outcomes. The study results show that teachers had some confidence that the transformations and feedback were appropriate for unseen current and future incorrect submissions in the cluster. However, we have not shown whether students find the feedback relevant to them or whether it improves their learning outcomes. There may be a trade-off between the generality and the relevance of the feedback teachers provide. Future studies can shed light onto how propagated feedback impacts student learning and inform how the systems could best help teachers to write feedback that is both general and pedagogically useful.

So far, our systems have only been shown to propagate feedback for small fixes. For MISTAKEBROWSER, this is due to the constraints of the training data: to build the training examples, we used a correct solution paired with the last incorrect submission from each student. It may be possible to synthesize larger fixes by learning transformations from correct solutions paired with incorrect submissions selected from earlier attempts in a student's submission history. While larger fixes may allow teachers to give feedback on more problems, there is an inherent tradeoff: larger fixes may be harder for teachers to understand and provide feedback for.

Our datasets have thus far only focused on fixing short functions typical of early exercises in an introductory programming class. The student solutions usually consist of just one main function, sometimes including a few helper functions. We have not tested how the systems' real-time performance will scale with more complex programs. Intuitively, the time to synthesize a fix for a given incorrect submission will depend on transformation size, the size of the incorrect submission, and the runtime of the test cases. The synthesis back-end is capable of learning and applying transformations for complex code bases (150K-1500K lines of code) [20], but tuning may be necessary to synthesize fixes for complex programs.

CONCLUSIONS AND FUTURE WORK

We presented two mixed-initiative systems for providing reusable feedback at scale with program synthesis. MISTAKEBROWSER learns transformations to fix incorrect submissions from examples of student-written bug fixes. MISTAKEBROWSER uses these transformations to cluster incorrect student submissions. Teachers can then review these clusters and write reusable feedback for current and future incorrect submissions. When examples of student fixes are not available, FIXPROPAGATOR allows teachers to write example bug fixes themselves. The system then learns from such fixes in real-time. We conducted two user studies with teaching assistants to evaluate our systems. Our results suggest that synthesized fixes, either from teachers' examples or previous students' bug fixes, can be useful for providing reusable feedback at scale.

As future work, we plan to deploy these systems in a massive programming course and evaluate the effectiveness of the generated feedback on helping students during programming assignments. To increase flexibility, we plan to combine workflows of FIXPROPAGATOR and MISTAKEBROWSER, allowing teachers to edit transformations learned from students' fixes by providing additional examples. To improve the interpretability of learned transformations, we plan to explore alternate visual and natural language interfaces to help teachers understand and modify synthesized code transformations. Finally, we have not yet investigated how to effectively combine both teacher-authored feedback and automatically synthesized hints in a student-facing interface. In future work, we plan to explore this design space of hybrid hints.

ACKNOWLEDGMENTS

We would like to thank the CS61a teaching staff for their time and perspective. 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, and a Google CS Capacity Award.

REFERENCES

1. 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.
2. Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? *ACM-TOCHI* 22, 2 (2015), 1–24.
3. Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of CHI*. ACM, 2491–2500.
4. Elena L. Glassman, Lyla Fischer, Jeremy Scott, and Robert C. Miller. 2015. Foobaz: Variable Name Feedback for Student Code at Scale. In *Proceedings of UIST*. ACM, 609–617.
5. Elena L. Glassman, Aaron Lin, Carrie J. Cai, and Robert C. Miller. 2016. Learnersourcing Personalized Hints. In *Proceedings of CSCW*. ACM, 1626–1636.
6. 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-TOCHI* 22, 2 (2015), 1–35.
7. Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the Symposium on Principles and Practice of Declarative Programming*. ACM, 13–24.
8. Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of UIST*. ACM, 599–608.
9. Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the CHI*. ACM, 1019–1028.
10. Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *Proceedings of the First Annual Workshop on Massive Open Online Courses*. 25–32.
11. Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. In *Proceedings of FSE*. ACM, 739–750.
12. Chinmay E Kulkarni, Michael S Bernstein, and Scott R Klemmer. 2015. PeerStudio: rapid peer feedback emphasizes revision and improves performance. In *Proceedings of L@S*. ACM, 75–84.
13. Timotej Lazar and Ivan Bratko. 2014. Data-driven program synthesis for hint generation in programming tutors. In *International Conference on Intelligent Tutoring Systems*. Springer, 306–311.
14. Joseph Bahman Moghadam, Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. 2015. AutoStyle: Toward Coding Style Feedback at Scale. In *Proceedings of L@S*. ACM, 261–266.
15. Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of WWW*. ACM, 491–502.
16. Donald A. Norman and Stephen W. Draper. 1986. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates, Inc.
17. Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of ICML*. IMLS, 1093–1102.
18. Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of OOPSLA*. ACM, 107–126.
19. Kelly Rivers and Kenneth R. Koedinger. 2015. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *IJAIED* (2015), 1–28.
20. Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of ICSE*. IEEE, in press.
21. Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question Independent Grading Using Machine Learning: The Case of Computer Program Grading. In *Proceedings of KDD*. ACM, 263–272.
22. Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.
23. Kurt Vanlehn. 2006. The behavior of tutoring systems. *IJAIED* 16, 3 (2006), 227–265.