Towards Answering "Am I on the Right Track?" Automatically using Program Synthesis

Molly Q Feldman
Cornell University
USA
molly@cs.cornell.edu

Yiting Wang
Cornell University
USA

William E. Byrd
University of Alabama at Birmingham
USA

François Guimbretière
Cornell University
USA

Erik Andersen
Cornell University
USA

Abstract

Students learning to program often need help completing assignments and understanding why their code does not work as they expect it to. One common place where they seek such help is at teaching assistant office hours. We found that teaching assistants in introductory programming (CS1) courses frequently answer some variant of the question "Am I on the Right Track?". The goal of this work is to develop an automated tool that provides similar feedback for students in real-time from within an IDE as they are writing their program. Existing automated tools lack the generality that we seek, often assuming a single approach to a problem, using hand-coded error models, or applying sample fixes from other students. In this paper, we explore the use of program synthesis to provide less constrained automated answers to "Am I on the Right Track" (AIORT) questions. We describe an observational study of TA-student interactions that supports targeting AIORT questions, as well as the development of and design considerations behind a prototype integrated development environment (IDE). The IDE uses an existing program synthesis engine to determine if a student is on the right track and we present pilot user studies of its use.

CCS Concepts • **Applied computing** → **Education**; • **Software and its engineering** → *Automatic programming*.

Keywords Computer science education, program synthesis, user interfaces

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10…$15.00
https://doi.org/10.1145/3358711.3361626

ACM Reference Format:
Molly Q Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards Answering "Am I on the Right Track?" Automatically using Program Synthesis. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3358711.3361626

1 Introduction

Personalized feedback contributes significantly to the success of learning experiences for novice programmers. To obtain personalized feedback, novices seek out help from numerous sources: peers, online references and, in the case of university students, professors, teaching assistants (TAs), or tutors. Yet the supply of help from these sources, who we call "feedback providers," frequently cannot keep up with the demand. For example, there are over 5 million unanswered questions on StackOverflow as of August 2019.[1] In traditional education, hiring in CS departments cannot keep up with skyrocketing enrollment.[2] In order to meet the demand for personalized feedback, we need to look outside of typical one-on-one help scenarios and towards new approaches. Our vision is to provide personalized feedback automatically, while maintaining the quality of the classic help experience.

Existing feedback technology generally falls into two categories: tools that augment existing feedback providers and fully automated methods. Augmentation takes the form of summarization tools that allow feedback providers to do their job more efficiently or systems that facilitate peer grading [28]. Recent work on providing feedback at scale considers in-progress code review [31], visualizing code execution [26], or facilitating one-to-many feedback sessions via an online chat interface [21]. However, most of these tools address a student's need for feedback indirectly through a feedback provider. They provide the ability to productively scale up human work, but are limited by the number of peers or TAs

[1] http://web.archive.org/web/20190828041932/https://stackoverflow.com/unanswered
[2] https://www.nytimes.com/2019/01/24/technology/computer-science-courses-college.html

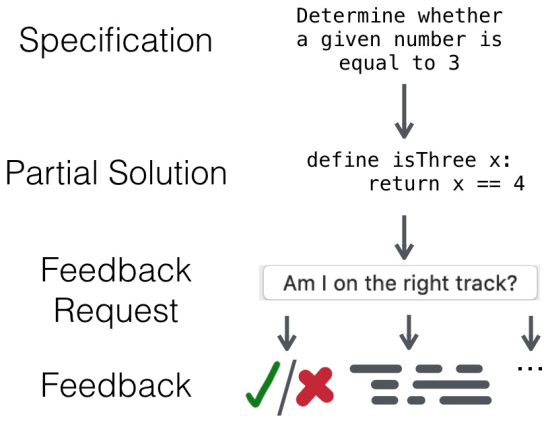

Figure 1. Feedback flow for an ideal automated feedback tool. First, the student sees the specification, then they can write a partial solution, and finally ask for feedback. The feedback can take many final forms, including YES/NO feedback and providing a partial or full solution directly to the student.

available. In comparison, our approach looks at helping the student directly.

In comparison to augmentation methods, automated tools attempt to mimic human capabilities. Past work in this area typically requires embedding some form of prior information about how to provide feedback into the system. This includes using data collected from previous or current students [22, 33] or modeling candidate incorrect solutions via an error model [35]. By using this information, these tools are able to provide a wide variety of meaningful feedback automatically to students. However, they cannot fully generalize to never-before-seen student errors or, in some cases, new assignments. We aim to limit the prior information needed to the language, the program specification, and a set of representative test cases. Succeeding at this approach would mean we can build automated systems that require very little human effort (no updates to data or error models) and scale to many different learners and learning environments.

This work builds on three key principles, guided in part by a study of TA-student interactions described in the first part of this paper. First, feedback should be integrated into the student's programming process. Ideally, the student writes their program, asks for feedback, and receives it, all in the same environment. Second, feedback should be provided in real-time, on request, and nearly instantaneously. The time of day or number of students in a room should no longer be a bottleneck. Finally, since there are many ways to solve any given problem, the system should be general enough to capture all solution strategies, rather than a single solution process specified by a reference solution.

Based in part on our preliminary study, this work focuses on a specific question: Can we use program synthesis to build a tool that can answer "Am I On the Right Track" questions and present the answer in a meaningful way to users? To answer this overarching question, we focus on the interplay between generating feedback and the user's interaction with the feedback tool. The type of interaction we envision is shown in Fig. 1. Given a specification for an assignment, students can write partial solutions for `isThree`. If they are not confident in their progress, they can request feedback. The tool will then provide feedback based on whether they are on the right track towards a correct solution. Our TA study suggests that this kind of feedback, which we call "Am I on the Right Track?" (AIORT) feedback, is common in office hours. Students can then revise their current solution or, if they were on a correct solution path, continue programming.

A motivating objective of this work is to leverage existing program synthesis infrastructure for the feedback domain. We use a program synthesis engine to generate feedback by synthesizing program fixes between the student's current implementation and a correct implementation, using an adapted sketching approach. This means we can theoretically capture a wide range of solutions to a given problem, rather than referencing known student errors. This general approach can be real-time and scalable, as recent advances in program synthesis make engine runs take only seconds. We show that it is possible to use an "off-the-shelf" engine, rather than a purpose-built solution, for educational applications. We are also in the process of developing a prototype integrated development environment (IDE) that allows us to convey the synthesized feedback to a user. We evaluated the IDE throughout our iterative design process, leading to both a final design plan and some recommendations for others considering broader use cases of existing synthesis engines.

2 TA Feedback in Office Hours

To design an automated feedback tool, we decided to gain insight into what functionality it should support by exploring the type of help feedback providers provide. Like many programming educators, we, the authors, have some sense of how we provide feedback to students. However, this is primarily determined by our own experience. Some educators have furthered their knowledge of feedback, studying feedback from the instructor's perspective [15, 40] and in the laboratory setting [4]. In comparison, we wanted to understand TA-student interactions in a realistic environment as, by and large, TAs are the primary feedback providers for CS1. Therefore, we developed a formative, observational study of office hours to understand how TAs provide feedback. We were partly motivated to conduct this study by existing work using formative studies for programming system design [7].

2.1 Study Design and Mechanics

Our formative study focused on observing TAs during their office hours for two Introduction to Computer Science courses at our institution, which we will refer to as CS1A and CS1B. CS1A was a Summer 2017 CS1 course taught in Python with 74 enrolled students, 7 TAs, and one professor. All TAs had

previous experience as teaching assistants in computer science at our institution. CS1B was a Spring 2018 CS1 course taught in MATLAB with approximately 250 enrolled students, 6 graduate TAs (1 new), 27 undergraduate TAs, and the same professor. The two courses cover the same general curriculum, with some changes due to the length of the summer session and the programming language. The topics covered by both courses include basic programming features, objects, recursion, and `for`/`while` loops. Our aim in studying these courses was to diversify our understanding beyond a single beginning language or class structure.

We designed the study to be as unobtrusive as possible, since our primary goal was to capture the genuine nature of office hours. We therefore recorded TA-student interactions with the medium chosen to minimize intrusiveness in different settings. The mediums included handwritten or typed notes, audio, or video recording. For CS1B, we shadowed only graduate student office hours, as the professor was concerned with potential overcrowding in undergraduate sessions. TAs and students were asked at the beginning of the study if they would like to participate in a voluntary research study focused on understanding how TAs provide feedback. If their consent was obtained, it covered the length of the study period (the last month of each course). Although shadowing office hours allowed us to observe TAs' interactions with students, we augmented our observational study with optional, unscripted interviews with the TAs. The interviews allowed us to obtain a perspective on TAs' impressions of effective feedback practices.

2.2 Data Analysis

We collected 37 TA-student interactions from CS1A and 13 from CS1B. Five of the CS1A interactions were excluded for various reasons,[3] leaving us with a total of 32 interactions for CS1A. We transcribed the usable interactions and then performed a grounded theory analysis [8].[4]

Our analysis aimed to characterize the type of help the TAs provided. We performed our initial analysis on the CS1A course data using three individual coders. The first coder (first author) assessed all of the interactions, developed a set of categories for types of TA-student interactions, and classified all 32 data points. A second coder (second author) provided example interactions for these categories without looking at the data. The second coder and a third coder (research intern) then independently categorized the interactions into the existing categories. We calculated a Fleiss' Kappa agreement score for each independent category and received results ranging from slight to almost perfect agreement, with on average moderate agreement. As an example, the score was 0.65 for Salvage. Given the full cycle of analysis for the CS1A data, the first author solely coded the subsequently collected CS1B data. All selected quotes below have been lightly edited and the participants anonymized (TA1A - TA7A for CS1A and TA1B - TA6B for CS1B).

2.3 Results

Our study suggests that TAs provide feedback aimed at helping students reach a correct solution and that feedback frequently occurs at different levels of granularity. Our grounded theory analysis identified 7 categories of feedback that range from Specification, in which the TA explains the assignments' specification to the student, to Language Feature, where a TA explains what functions to call to accomplish a certain task (Fig. 2). An example of Language Feature feedback would be TA5A's feedback *"*`randint` *needs a minimum and a maximum value"* when helping a student understand the input to a library function.

In contrast to the finer-grained feedback, feedback in the Salvage category is more holistic. When a TA begins salvaging a student's program, they start out by identifying an error in the student's current solution ("traditional" debugging). Then they ask guiding questions to determine the student's confusion, assess the student's conceptual understanding, and ultimately lead the student to identify the error(s) themselves. This guidance during the student's solution process is what specifically differentiates salvage from other types of feedback. Of particular note is how the TAs provide the feedback. They do not tend to instruct the student to start over from scratch, but rather to make local changes (TA4A: *"I think we can simplify a little bit and change the order of the recursive call"*) that may then have far-reaching implications. For instance, in the same interaction with a student, TA4A ended up guiding the student from a solution containing a `for` loop and an accumulator to a correct recursive solution.

In our 5 interviews with CS1A and CS1B TAs, they spoke specifically about paths to correct solutions as a core consideration behind how they provide feedback. They discussed how to identify if a student's work is correct, when (or if) to present a fix, and where in the solution process they provide help. TA1A made the following general points:

> *Usually I'll try and figure out what's wrong, and then I'll give them, like a hint or something, about how to get there ... if it's something really weird, I'll just straight up tell them "your implementation here is not right."*

TA1B discussed what to do when a student is too far down an incorrect path for their work to be salvageable:

> *When I run out of ways to help them, I sometimes show them the answer. I show them "this is the way to do this"*

[3] The five interactions were excluded either because of errors with recording (2) or prominent unconsented student voices (3)

[4] Grounded theory analysis contends that we can understand the content in communication by analyzing the type of "common ground" shared by the speakers. This standard qualitative analysis method, like most qualitative research, relies on two or more "coders" who categorize interactions according to the given property being studied and the research goals. These coders then typically reach consensus about each data point's final classification.

> *and I try to explain backwards. Like, "knowing this is the correct solution, how would you get here?" ... Even though I give them the answer, I hope they understand the process.*

TA3A considered the situation of a student beginning with a general conceptual approach to a problem and narrowing it down to a specific implementation:

> *Usually they will get to the point where [they will say] "I probably should be using some kind of iteration" and then we can talk about ... if they're on the right track, and if they're not, I'll give them more guidance to steer them in the right direction. But I like to have them at least have some trial and error in office hours, if I can.*

Taken together, our interaction classification and our TA interviews suggest a theme. Most students want an answer to some version of the question "Am I on the Right Track?" and most TAs want to provide some version of an answer to that question. The help they provide varies and can be TA-specific; it can be conceptual, simply a "Yes" or "No" answer, or even the entire solution to the problem.

To be clear, we are not making a broad claim about a pervasive existence of this classification and theme. While on the one hand the study targeted two courses with different programming languages, student bodies, and TAs, the amount of data was rather limited and reflected a narrow population. For example, office hours for CS1B graduate TAs were sparsely attended and, as mentioned above, we were not permitted to observe the undergraduate office hours for CS1B, where we believe that the majority of interactions occur. This study also took place in a limited time period (the last month of material for each course) and we only studied courses at a single institution. We also approached this study with the idea of building an automated feedback tool in mind. Nonetheless, the study provides evidence of the potential value of "Am I on the Right Track?" feedback in particular settings.

3 Synthesizing Feedback Automatically

In this section, we motivate why and how one might pose "Am I on the Right Track?" (AIORT) feedback as a synthesis query. AIORT feedback relies on determining whether a student can arrive at a correct solution, given their current partial solution. Formally, we want to determine if there exists a set of local changes that can be applied to the student's partial program P_S to transform it into a final program that has the same "functionality" as a desired correct solution.

Local changes to a student's partial solution are captured by the addition and deletion of code. For instance, for the following incorrect implementation of `isThree`, it is clear that the 4 should be deleted and a 3 added in its place:

```
define isThree x:
    return x == 4
```

Feedback	The TA...
CS Concept	explains a high level programming design feature (e.g. recursion, `while` loop)
Grading	answers a student question about an assignment or exam grading
Guidance	provides high level guidance about an approach to a given problem
Language Feature	explains how to use a language feature (e.g. library function)
Salvage	helps the student towards a correct solution path, given their current (typically incorrect) code
Specification	explains a function specification (e.g. the overall goal, in scope or out of scope inputs, etc.)
Test Case	suggests a test case for the student to consider
Type	helps the student by highlighting type information (e.g. input types for functions or methods)

Figure 2. Classification concepts for TA-student interactions.

However, consider this incorrect implementation of `replaceEight`, a function that, when implemented correctly, recursively replaces all elements in a list with the number 8:

```
define replaceEight x:
    if empty x:
        return [8]
    else:
        return 8 ++ replaceEight (rest x)
```

This implementation is possibly incorrect in either the base case condition or its return value, depending on the wording of the specification. In the first case, the condition should be changed to `len x == 1` and, in the second case, the return value for the base case should be changed to the empty list. This is an example where there are multiple local changes that can result in a correct solution, but it is not clear which one to choose.

Furthermore, for any reasonable definition of local, there are an infinite set of possible changes, which makes it difficult to find the "right" one, even if we could formally specify such a notion. Yet, we would like to provide a general solution to this problem in order to generate AIORT feedback. The number of possible changes makes many simple algorithms, such as brute force search, infeasible. However, program synthesis is frequently used to automatically generate code edits towards some specification in an infinite search space [19]. This is the approach we take.

Modern program synthesis algorithms tend to use *sketching*, with some notable exceptions (e.g. Synquid [30]). Introduced by StreamBit [37] and made popular by SKETCH [36], sketches are partial implementations with *holes* that denote where the engine should synthesize new code. Holes

are written "??" as seen in the following sketch for `isThree`:

```
define isThree x:
    return x == ??
```

AIORT feedback is a non-standard use case for sketches because of the frequency and variability of the queries. Usually users either create a single sketch or create a mechanism for generating similar sketches automatically at pre-determined intervals. In contrast, AIORT feedback needs to be generated in real-time and repeatedly at variable intervals.

Next, we consider how to generate sketches based on a student's partial solution. We take as input a correct implementation P_I, typically provided by an instructor, a set of representative test cases, and the student's partial solution P_S. From P_S, we attempt to synthesize a program that matches the output of P_I on all representative test cases.

Since we are building a user-facing tool, we need to consider whether to expose the notion of holes directly to the tool's user. Although this is typical in most synthesis applications, we believe that it is ill-advised for educational use cases. On the one hand, intuitively, exposing holes would seem to allow users to specify holes exactly where they believe code should be added, potentially instilling more understanding and confidence in the synthesized feedback. However, from pilot studies with TAs, we found that because most users are not accustomed to interacting with synthesis tools, explicitly surfacing holes appears to be confusing and potentially detrimental to the feedback process. We instead insert holes automatically without any user intervention. We developed two main principles for automatic hole insertion based on our feedback goals: (1) we only consider adding, not deleting, code and (2) we do not add holes as inner nodes of the AST.

The idea behind not deleting code is being able to provide clear, correct, and contextualized feedback. First, giving feedback in the context of their code helps students learn. Maintaining all of the code they wrote and only considering additions allows for the most context possible. Second, standard sketches provide no facility for deleting code. To delete code, we would need to choose a piece of the student's partial implementation, delete it, and replace it with a hole. However, the above `replaceEight` example shows that which code deletion to choose can be non-obvious without user input. In this model, we would have to perform deletions automatically, which could result in synthesized output that differs significantly from what the student intended.

In general, holes can be added in any location in the student's code. This does include degenerate cases, notably "expression wrapping" in which a given expression e can be turned into an argument of a hole (e.g. `??` e). There are numerous issues with allowing wrapping. First, for a general synthesis approach, the algorithm can consider all functions in the source language, including those that the student has not previously seen. Another problematic synthesis result is a "guarded false" statement (e.g. `if false:`), which provides no feedback on the student's work that has been wrapped.

Program	P	::=	$D\ D^* \ldots$
Definition	D	::=	`(define` $x\ e$`)`
Expression	e	::=	$x \mid c \mid p \mid \ell \mid$ `'`$d \mid$ `(quote` d`)`
		$\mid$	`(lambda` $f\ e$`)` $\mid$ `(if` $e_1\ e_2\ e_3$`)`
		$\mid$	`(` $e\ e^* \ldots$ `)`
Formals	f	::=	$x \mid$ `(` $x^* \ldots$ `)`
Constant	c	::=	`#t` $\mid$ `#f` $\mid n \mid s$
Datum	d	::=	$c \mid a \mid$ `()`
Predicate	p	::=	`and` $\mid$ `or` $\mid$ `not` $\mid$ `<`
		$\mid$	`>` $\mid$ `equal?` $\mid$ `null?`
ListOp	ℓ	::=	`cons` $\mid$ `car` $\mid$ `cdr` $\mid$ `list`

Figure 3. Grammar for the subset of Scheme supported by our tool (adapted from [1]). Here x is a legal Scheme variable, n is a legal Scheme number, a is a legal Scheme symbol, s is a legal Scheme string, and the notation $g^* \ldots$ represents zero or more occurrences of syntactic entity g. Although users can use `cons` to make pairs of datums and `list` to form lists, we do not support feedback for either pairs or non-`cons` lists.

To exclude these cases, we do not add holes as inner nodes of the AST, instead only inserting holes as leaves (e.g. in place of arguments missing from a function). We require a map from functions to their arity, from which we can calculate the necessary number of holes to insert.

To exhibit our principles, consider the following partial implementation of `isThree`:

```
define isThree x:
      return  == (1 + )
```

Below are two conceivable ways of adding holes to this implementation:

```
define isThree x:          | define isThree x:
 return ?? == (1 + ??)     |  return ?? (?? == ??)
```

Our approach would produce the sketch on the left, but not the one on the right. In order to produce the one on the right, we would need to break both of our principles: (1) delete student code (i.e. `(1 + )`), replacing it with a hole and (2) insert a hole as an inner node, producing the bold **??**.

3.1 Implementation

To explore the viability of our approach via a prototype tool, we instantiated it using an existing synthesis engine with a sketching-style interface. There are numerous such engines publicly available: Rosette [39], Barliman [6], SKETCH and its derivatives [36], methods using the PROSE SDK [32], among others. Two characteristics of an ideal synthesis engine for our use case are language choice and generality. Some methods are written in a base language and allow the user to specify their synthesis query via a DSL whereas others directly support a specific subset of an existing language. Ultimately, we wanted the engine to support a language used extensively by novices, with input assumptions that are not more restrictive than the requirements discussed above.

Our prototype uses the Barliman engine [6]. Barliman synthesizes programs in Scheme using the miniKanren logic programming language. Barliman's use of Scheme is well-suited to an educational environment, as Scheme-like languages are commonly used for CS1 courses (e.g. Brown University [25] and Indiana University [34]) or as a first functional language. Scheme-style languages are also likely to be unfamiliar to non-novice programmers learning a new language, presenting another student population for evaluation. The specific subset of Scheme our tool supports is shown in Fig. 3. We chose the subset of Scheme based on both Racket Beginning Student Language [12] and the Scheme subset supported by Barliman.

Runs of Barliman adapted for our tool can either fail or succeed. When Barliman fails, it means that, given a set of input test cases and a partially completed program with holes, there are no Scheme expressions that can fill the holes to produce a correct solution. If Barliman fails on all of our sketches of the student's solution, then we say that they are not on the right track (AIORT returns false). Any program that is either unparsable or not in our subset of Scheme (Fig. 3) will not be on the right track. Success means that Barliman has found Scheme expressions that produce a correct solution. In the event of a success, we say that the student is on the right track.

4 User Interface

As noted previously, our goal is to incorporate "Am I on the Right Track?" (AIORT) feedback into a student's learning environment. We chose to do this by tightly coupling the steps a student takes to write their program with how they receive feedback automatically inside a prototype integrated development environment (IDE) based on the principles in the previous section. The student can begin a programming assignment by first considering the specification and type signature of the goal function (Fig. 4, #2). They can then begin programming normally in the center text box of our IDE. However, our prototype currently only supports defining a single function per instance. This is a limitation of the prototype, not the approach. The rest of the IDE was designed to help facilitate the student's programming process. If the student forgets some syntax or semantics of a Scheme operator, they can use the Term Lookup section (#1) to find that information. To draw attention to potentially useful operators, we present results used in the instructor's reference solution at the top of the list, followed by the rest of the relevant operators that we support. As they approach a full solution, students can write tests and run them in the Test Cases section (#5). If students are not sure what tests to consider, they can ask for a generated test case at #4 and the tool will generate a test that passes the instructor's solution, but fails the student's solution. We provide generated tests as both an extra feature that can be synthesized and to provide a lower barrier to entry for success in a test-drive development process. Finally, if the student wants to restart the assignment, the arrow at #2 resets the center text box to the starting state.

The AIORT feedback feature is shown at #3. It functions as proposed in Fig. 1: the student can press the "Am I on the Right Track?" button at any time to receive a YES/NO answer. We transform the student's solution in the main text box to a sketch at the time the student presses the AIORT

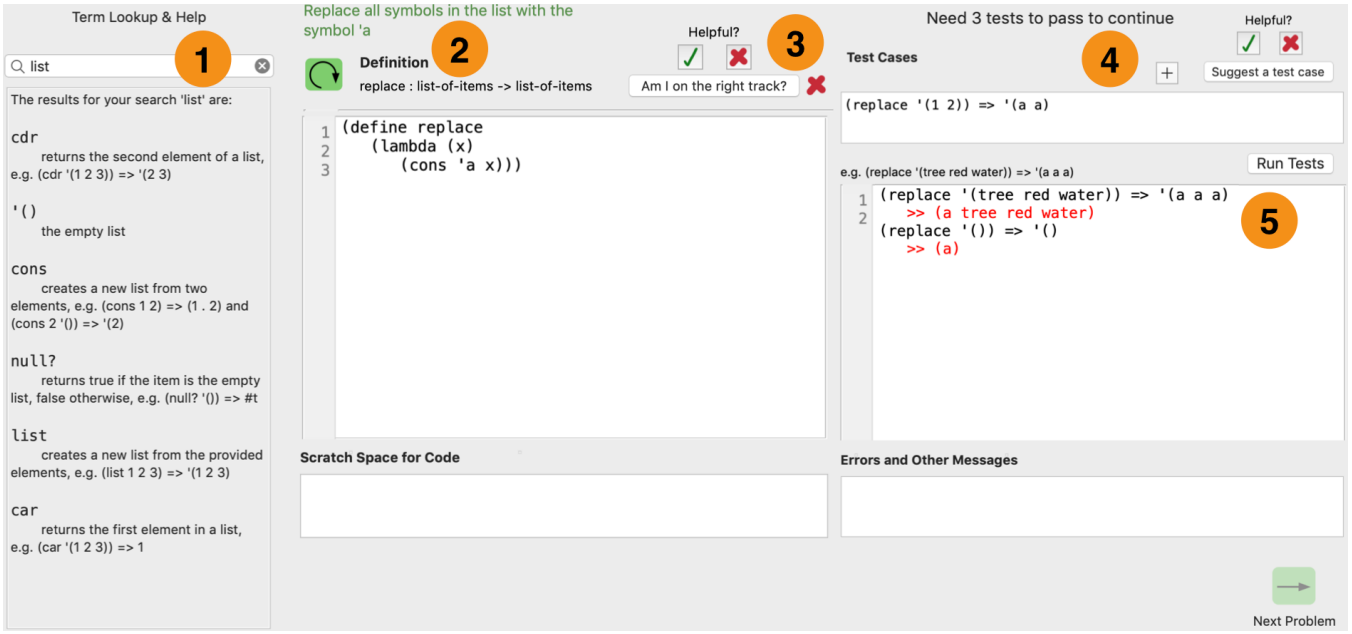

Figure 4. The current user interface for providing AIORT feedback; evaluated in the pilot study with novice Schemers.

button. We then execute our sketch generation strategy, run the resulting Barliman query, and, depending on the output, serve a YES, NO or MAYBE answer. MAYBE means that the Barliman query took too long to run, triggering a timeout that we added to the UI.

Our prototype interface is simple, as it was designed to help us iterate on how to present feedback to the user. This allows us to directly observe how users interact with AIORT feedback, especially as it differs significantly from standard IDE feedback options (e.g. linting or term autocomplete). In addition, a prototype allows us to control the learning environment and iterate quickly on different interaction styles. In the future, adding synthesized feedback to established educational IDEs (e.g. DrRacket [13]) would be advantageous.

5 Evaluation

We evaluated the performance of our approach, aiming to answer three main questions:

- Is our synthesis approach able to provide feedback on partial student solutions?
- Do users find the "Am I on the Right Track?" (AIORT) feedback helpful?
- How do users interact and engage with our tool?

First, we present some examples of the feedback our synthesis approach provides on real partial implementations. We then present two evaluations of our user interface. As noted above, we performed an iterative design of the UI in order to consider how users should interact directly with synthesized feedback. The first pilot user study considers an alpha version of the UI with TAs as our users and the second pilot study evaluates the UI described in detail above with students as our users.

5.1 Can We Synthesize Feedback?

During our pilot studies, we were able to explore how well our sketching approach generates feedback for users writing code in our prototype IDE. Here we provide some examples of what our system can synthesize. We did not collect extensive summary statistics or timing information, but in future work we would like to further test the robustness of our approach.

A key motivation for using a general purpose synthesis engine is supporting alternative correct solutions. For the function `first-not-doll?`, adapted from How to Design Programs (HtDP) [11], a student in our second pilot study wrote the following code, which the tool correctly said was on the right track:

```
(define first-not-doll?
    (lambda (x)
        (if (equal? 'doll (car x)) #f #t)))
```

Notice that, though this is a common and correct solution, it is likely not the instructor's solution provided to our system nor the typical solution an instructor might provide.

We also want to provide feedback in real-time during the solution process. In our first pilot study, we used a function `replace` (adapted from HtDP) which mimics our running example `replaceEight` above, but replaces elements with `'a` rather than `8`. One TA had the following partial solution:

```
(define replace
    (lambda (x)
        (if (null? x) '() (cons ))))
```

The synthesis engine produced the outputs `'a` and `(replace (cdr x))` to fill the two holes added automatically after `cons`. This is a significant component of the solution showing that our tool can support the beginning of the solution process.

In our studies, the synthesis engine did not always perform as the user would expect (see below). Anecdotally, we generally observed users disagreeing with the tool's synthesized feedback in one of three cases: (1) the engine said NO correctly and the user did not understand why, (2) we reached a UI-instituted timeout for Barliman and the user received no feedback, or (3) the user's internal model for whether they were on the right track did not match the system's model.

5.2 Code Snippet Feedback with TAs

Our first pilot evaluation focused on understanding TAs' perspectives on interacting with synthesis features, holes, and our IDE. The IDE and the synthesis features were in early stages when we performed this evaluation. Thus there were a few issues that impacted the tool and caused it to crash. The system evolved between the first and second pilot studies, but the overall approach remained the same.

Study Mechanics: We recruited 10 graduate students at our institution who each had been a TA for at least one CS course at a 4-year college or university. TAs were first presented with a consent form to sign and then were randomly assigned to either an A or B condition. In the A condition, the feedback feature was visible in the IDE and, in the B condition, it was not. We then provided all TAs with a short Scheme lesson via a pre-recorded video. After the Scheme tutorial, TAs were asked to replicate their practices for office hours by interacting with incorrect student code using our interface. We explained the basic interface functionality to the TAs, but did not explain the synthesis feature. This was a choice in order to capture the TAs' first impressions. We obtained the incorrect student solutions through a small study that we performed with novice functional programmers based on exercises in HtDP. After approximately 20 minutes of interacting with the tool we asked the TA to stop, fill out a post survey, and participate in a short interview about their experience with the tool. The session took about an hour.

Main Differences from current UI: There were two specific differences between the UI presented above and the alpha interface used in this evaluation (Fig. 5). First, the alpha interface exposed holes to the user. They were allowed to insert holes themselves and, when a hole was added automatically,

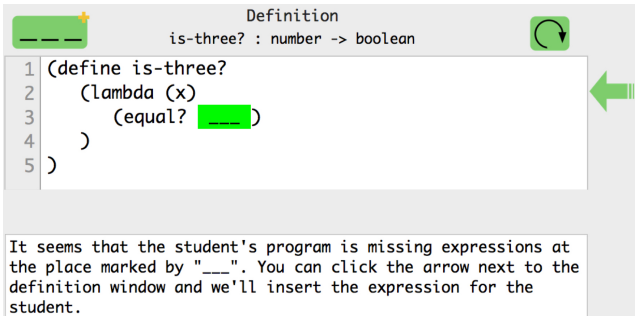

Figure 5. The central component of the early stage (alpha) interface used in the TA pilot study. If the user pressed the green arrow, `x 3` would be inserted at the hole marked `___`.

it was inserted in the code as `___`. If there were multiple holes next to each other (e.g. `(equal? ?? ??)`), we condensed them visually into a single hole. This choice was made to allow for some visual consistency; a hole means missing code. At the time, we also did not want to explicitly tell the student how many expressions were missing, because we think that they should learn to extract this information from the function themselves. Calls to create Barliman queries were also generated automatically every few seconds, rather than waiting for a button press. Second, after a Barliman run, the TA had the option to insert the fully generated code edit into their solution. We instantiated any logic variables by picking from a set of values of their type.[5]

Results: The results of our study were mixed, but suggested specific areas for improvement. One limitation was that the TAs did not experience many interactions with the AIORT feedback; out of 10 TAs, 6 of which were in the synthesis (A) condition, only 1 ended up seeing the output of the engine. Four of the six had holes automatically inserted, but they did not complete the code insertion process.

In general, the TAs had some negative comments about the feedback and how it was presented. For instance, one TA said that they felt that the fully automated feedback features caused them to have a lack of control over the process:

> *I feel like it is going ahead without me ... why is it going off and doing things! I don't want it to do things yet; it's stealing my thunder. Is it actually changing the code? I think it is which is scary cause I want the student to fix the code not the program. (TA #3)*

However, some TAs did see potential for the general approach of the tool. The same TA that complained about lack of control also noted:

> *I am not there and they are stumped, then having that button could on the one hand help them. But, on the other hand, just giving them the answer without forcing them to know why [is not good]. If they are stumped and diligent, they can ... be given the right answer and then go reverse engineer why that's the right answer and that's potentially really helpful to get somebody unstuck. (TA #3)*

This is similar to the feedback style of TA1B from our observational study.

Our major takeaway from this study was that exposing holes and the synthesized code was not the right interaction modality. This seemed particularly detrimental with TAs as our user population, given that they have both their own solution styles when writing code and specific ways in which they like to help students. We therefore chose to redesign the tool with opt-in, manual feedback features and relegated holes to a backend-only feature. This feedback, obtained from the early prototype of our interface (Fig. 5), drove us to change the interface to the one presented in Section 4.

5.3 YES/NO Feedback with Scheme Novices

Given the results of our TA pilot study, we performed another round of user interface design and obtained the design described in detail in Section 4. The goal of the second pilot study was to determine if the new interface is helpful and engaging. We recruited students with a background in programming, but no formal computer science education, from a group of graduate student scientists at our institution working in computation-adjacent fields (biology, social science, etc.). In total, five students enrolled in the study, via either email solicitations or our institution's SONA system. Students met the study criteria if they met the recruitment criteria and had some background in high-level programming (e.g. R, web development), but were excluded from the study if they could easily answer beginner questions in a functional programming course or had significant knowledge of a functional language. We had four valid responses in total.

Of the four valid responses we obtained, two students interacted with the synthesis feature and two students interacted with a control UI. After completing the consent form, students were given a walkthrough of how to use the tool and a Scheme lesson on a blackboard. The lesson was pre-planned to cover certain topics, but varied slightly from student to student. After the walkthrough and lesson, students were left on their own to complete as many exercises as possible out of 5 adapted from HtDP (`is-three?`, `replace`, `first-not-doll?`, `flip`, and `filter`). We required those who had the synthesis feature IDE to interact with the help features (AIORT feedback and test case generation) at least once per exercise as, in previous studies, participants tended to ignore them and proceed with programming as usual. We also implemented good/bad performance buttons that could be used to report opinions on AIORT feedback and test case generation (to the left of #3 in Fig. 4), although students were not required to use them. For the two students who had the

[5] We perform the same instantiation for automatically generated test cases

feedback UI, they reported helpfulness 10 times for AIORT feedback, including both positive and negative responses.[6]

Results: The main takeaway of our second study was a mismatch between our approach and the students' expectations. Of the 10 times the two students who experienced AIORT feedback reported performance, 8/10 times the performance report their impression was negative. However, we confirmed that the feedback was correct for all 10 instances and students were explicitly shown examples of how the feedback features worked. In general, the students had a lukewarm reaction to the tool: the students with feedback both responded "somewhat agree" to the statement "the feedback features were helpful when solving problems."

The mismatch between our model and student expectations seems to stem from how the tool determines if they are on the right track. As noted in our theoretical approach, we do not perform expression wrapping when inserting holes automatically. However, consider the following code from the study for `replace`:

```
(define replace
    (lambda (x)
        (cons 'a (replace (cdr x)))))
```

This implementation contains the correct recursive call for the `replace` function, but it is missing the base case. The feedback here is that the student is not on the right track because there are no AST leaves missing an expression. However, understandably, the student reacted negatively, since they believed they were on a correct solution path. Note that we believe that the negative reaction was to the exact response (YES/NO) from the feedback feature, not to the feature itself.

6 Discussion

A key takeaway is that, even when made explicit, there is a disconnect between how users perceive the synthesized feedback and its actual functionality, even when they believe that the tool is potentially helpful. We believe that this arises because working one-on-one with an automated feature as powerful, but equally as limited, as synthesis is a new experience for most coders. Therefore, our study results embolden us to explore how to better expose synthesis to the student.

A possible key improvement may lie in integrating the synthesized feedback better into the student's workflow. We are currently considering a different approach (Fig. 6) that

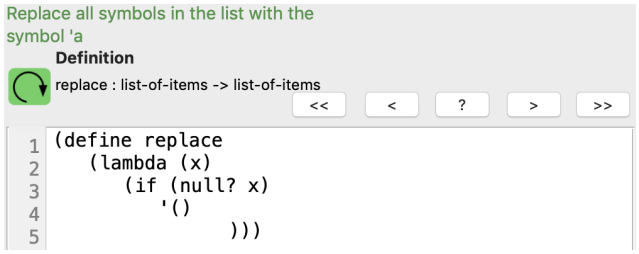

Figure 6. This new interface design allows users to request AIORT feedback with ? and revert to previous AIORT versions with << and <.

uses a user interface modeled after play/pause/rewind buttons. The goal is to allow the student to check if they are on the right track at any point (via ?) but also revert to previous checkpoints where they were on the right track. The << button replaces the current implementation with the last correct implementation obtained from pressing ?. The < button replaces the current implementation with the last correct implementation from a set of automatic AIORT checks performed every 2 seconds in the background. This button allows users to functionally "undo" the most recent change that lead them down an incorrect path. In theory, using all of ?, <<, and < as part of the programming process may help students learn how the synthesized feedback works in an intuitive way.

The main limitation of our evaluation was the size of the studies and their lack of robust quantitative analysis. The number of study participants for each interface iteration was small enough that we are unable to draw any statistically significant conclusions. As future work, it would be helpful to test our tool in a larger lab study or in a traditional Scheme learning environment. In particular, our user studies considered TAs and novice Schemers, but no CS1 students.

7 Related Work

7.1 Tools for Aiding Student Learning

Successful tools for CS1 manage to use student and staff effort more effectively. A classic example are educational IDEs or IDE plug-ins built to help scaffold learning, such as DrRacket [13] and BlueJ [27]. Some tools go beyond the coding process and specifically address test-driven development by using peers to assess and provide test cases [31]. There has been a study on extending a single staff member's effectiveness by allowing them to work with many students in real-time [21]. Peer grading systems structure the process of providing peer feedback for hundreds of students when there are staffing limitations [9, 28]. Other recent work has focused on visualization, such as modeling code execution [20, 26], student choices of variable names [16], and student solutions at scale [18]. Although these tools make better use of human effort, they have some limitations: (1) for systems that rely on peers, feedback quality can be variable and (2)

[6] Although not our main focus, we also obtained some results for evaluating automatic test case generation. There were 10 helpfulness ratings for test case generation (5 positive, 5 negative). Both students responded "somewhat disagree" to "The tool suggested test cases I would not have come up with on my own." This reaction may have occurred because the students were skilled enough with non-functional programming to write meaningful test cases. They responded 7/10 and 9/10 respectively to "On a scale of 1-10, how comfortable are you with programming?"

systems for staff members can improve productivity, but do not replace the benefits of additional personnel. HelpMeOut [22] is an approach that provides real-time, automated feedback to students learning Java for art and design applications. HelpMeOut leverages the use of existing knowledge about student errors to inform feedback for future students. Our approach does not consider other students' work.

Rooting the development of an automated feedback tool in human feedback practices is not a new concept; in fact, when writing on intelligent tutoring systems (ITS) for *Science* in 1985, Anderson et al. noted, "Computer systems ... are being developed to provide the student with the same instructional advantage that a sophisticated human tutor can provide" [2]. However, there is currently an incomplete understanding of what human tutor feedback looks like in office hours. There is work on how to increase diversity using TAs [29], how to use an apprenticeship model to train TAs [40], and what kinds of questions TAs ask [4, 15]. The goal of our TA study was to observe the full breadth of office hours.

7.2 Automated Feedback Techniques

Generating feedback for student work automatically has been studied since at least 1960 [24] by a number of different academic and industrial communities. ITS, which allow students to obtain real-time feedback while working on problems as part of a pre-specified curriculum, have been successful in domains such as K-12 math [5] and programming. The LISP tutor [3] has specific relevance to this work, as we use the Scheme programming language in our prototype implementation. Additional work has augmented ITS feedback by using real student data for Python programming [33]. Providing feedback without prior assignment-specific information is more difficult; recent work succeeded at synthesizing such feedback for K-8 math [10].

There is a current surge in work on automated feedback techniques for programming due to recent innovations in program synthesis. Singh et al. [35] can identify the smallest number of changes needed to transform an incorrect student Python program into a correct program using a hardcoded error model. Subsequent work by Head et al. [23] and Suzuki et al. [38] has been able to provide more general Python feedback based on the Refazer system, which synthesizes Python program transformations. Head et al. built two systems, one based on real student data, to cluster student code and provide bug fixes for buggy student programs in Python. Our system is most similar to FixPropogator, which iteratively improves bug fixes to student code based on teacher provided results. In contrast, our work focuses on replicating interactions from office hours in a functional programming context and in real-time. Ask-Elle also provides functional programming feedback, by guiding students towards model solution strategies in Haskell [14]; our work aims to support the breadth of individual student responses. In the data-driven feedback literature, work has looked at different interaction techniques for how to present help [17]. Specifically of relevance to our future work is their discussion of the "push" and "pull" models for providing help, as well as whether it is preferable to resolve a given error by guiding the student using the most common error fix or the most correct solution.

8 Conclusions

This paper explores how to provide "Am I on the Right Track?" (AIORT) feedback automatically for programming assignments using program synthesis. Our approach considers how to provide general feedback in real-time without using information about known student errors. To implement this idea, we considered a non-standard use case for automatically generating program synthesis sketches. We also performed an iterative, and ongoing, design process for an IDE which presents synthesized feedback to a user.

Applying state-of-the-art synthesis tools directly to education, rather than adapting or building purpose-built algorithms, is a challenging idea. We have some recommendations that we believe may set up others who want to take on this challenge for success. First, we have shown that synthesis *can* work in this limited information environment. This expands our understanding of where synthesis can be used to provide feedback. We also see understanding the educational context as key to the success of any synthesis-based feedback tools. Performing a formative study on the educational component you are trying to target, as we did in our TA study, can reinforce design decisions for the tool at large.

Ultimately, the main challenge of applying state-of-the-art PL algorithms to education is that the algorithms and the domain are not natively compatible. As seen in our approach section, it is difficult, and sometimes entirely unclear, how to obtain the right balance between the computational needs of the algorithms, the ideal education situation, and engaging user interaction. This is not a new problem and we are nowhere near the first to explore it. However, it is not going away, since synthesis continues to evolve and more novel applications domains are being considered. We propose that designs like ours take a step in the right direction for exploring how to apply computational systems to education.

Acknowledgements

Thanks to our study participants & Daisy Fan; Xinqi Lyu, Yuntian Lan & Jinyan Zheng for data analysis and coding; Haym Hirsh & Jonathan DiLorenzo for helping craft the paper; and Youyou Cong & Adrian Sampson for their PL perspectives. Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number 3OT2TR002517-01S1. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

[1] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, R Halstead, et al. 1998. Revised 5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* (1998).

[2] John R Anderson, C Franklin Boyle, and Brian J Reiser. 1985. Intelligent tutoring systems. *Science* 228, 4698 (1985), 456–462.

[3] John R Anderson, Frederick G Conrad, and Albert T Corbett. 1989. Skill acquisition and the LISP tutor. *Cognitive Science* 13, 4 (1989), 467–505.

[4] Kristy Elizabeth Boyer, William Lahti, Robert Phillips, Michael D Wallis, Mladen A Vouk, and James C Lester. 2010. Principles of asking effective questions during student problem solving. In *Proceedings of the 41st SIGCSE Technical Symposium*. ACM, 460–464.

[5] John Seely Brown and Richard R Burton. 1978. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive science* 2, 2 (1978), 155–192.

[6] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017).

[7] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, 963–975.

[8] Herbert H Clark, Susan E Brennan, et al. 1991. Grounding in communication. *Perspectives on socially shared cognition* 13, 1991 (1991), 127–149.

[9] Luca de Alfaro and Michael Shavlovsky. 2014. CrowdGrader: A tool for crowdsourcing the evaluation of homework assignments. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 415–420.

[10] Molly Q Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. 2018. Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM.

[11] Matthias Felleisen, Robert Bruce Finder, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to design programs: an introduction to programming and computing*. MIT Press.

[12] Matthias Felleisen, Robert Bruce Finder, Matthew Flatt, and Shriram Krishnamurthi. 2014. *How to Design Programs, Second Edition*. MIT Press.

[13] Robert Bruce Findler. 2014. DrRacket: The Racket Programming Environment. (2014).

[14] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2017. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 65–100.

[15] Michael Glass, Jung Hee Kim, Martha W Evens, Joel A Michael, and Allen A Rovick. 1999. Novice vs. expert tutors: A comparison of style. In *MAICS-99, Proceedings of the Tenth Midwest AI and Cognitive Science Conference*. 43–49.

[16] Elena L Glassman, Lyla Fischer, Jeremy Scott, and Robert C Miller. 2015. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 609–617.

[17] Elena L Glassman, Aaron Lin, Carrie J Cai, and Robert C Miller. 2016. Learnersourcing personalized hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. ACM, 1626–1636.

[18] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip Guo, and Robert C. Miller. 2015. OverCode: visualizing variation in student solutions to programming problems at scale. *Transactions on Computer-Human Interaction* (2015).

[19] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

[20] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 579–584.

[21] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 599–608.

[22] 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 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.

[23] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 89–98.

[24] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.

[25] John F Hughes. 2018. CS: An Integrated Introduction. http://cs.brown.edu/courses/cscio170/

[26] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, 737–745.

[27] Michael Kölling and John Rosenberg. 1996. An object-oriented program development environment for the first programming course. In *ACM SIGCSE Bulletin*, Vol. 28. ACM, 83–87.

[28] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R Klemmer. 2013. Peer and self assessment in massive online classes. *ACM Transactions on Computer-Human Interaction (TOCHI)* 20, 6 (2013), 33.

[29] Tia Newhall, Lisa Meeden, Andrew Danner, Ameet Soni, Frances Ruiz, and Richard Wicentowski. 2014. A support program for introductory CS courses that improves student performance and retains students from underrepresented groups. In *Proceedings of the 45th ACM SIGCSE*. ACM, 433–438.

[30] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[31] Joe Gibbs Politz, Joseph M Collard, Arjun Guha, Kathi Fisler, and Shriram Krishnamurthi. 2016. The Sweep: Essential Examples for In-Flow Peer Review. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 243–248.

[32] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 107–126.

[33] Kelly Rivers and Kenneth R Koedinger. 2014. Automating hint generation with solution space path construction. In *International Conference on Intelligent Tutoring Systems*. Springer, 329–339.

[34] Chung-chieh Shan and Robert Rose. 2019. C211/H211: Introduction to Computer Science. https://www.cs.indiana.edu/classes/c211/

[35] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.

[36] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching stencils. *ACM SIGPLAN Notices* 42, 6 (2007), 167–178.

[37] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 281–294.

[38] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, and Björn Antoni, Loris D'and Hartman. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE.

[39] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 135–152.

[40] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. 2013. Massive increase in eager TAs: Experiences from extreme apprenticeship-based CS1. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 123–128.