High-Coverage Hint Generation for Massive Courses

Do Automated Hints Help CS1 Students?

Phitchaya Mangpo Phothilimthana
University of California, Berkeley
mangpo@eecs.berkeley.edu

Sumukh Sridhara
University of California, Berkeley
sumukh@berkeley.edu

ABSTRACT

In massive programming courses, automated hint generation offers the promise of zero-cost, zero-latency assistance for students who are struggling to make progress on solving a program. While a more robust hint generation approach based on path construction requires tremendous engineering effort to build, another easier-to-build approach based on program mutations suffers from low coverage.

This paper describes a robust hint generation system that extends the coverage of the mutation-based approach using two complementary techniques. A syntax checker detects common syntax misconception errors in individual subexpressions to guide students to partial solutions that can be evaluated for the semantic correctness. A mutation-based approach is then used to generate hints for almost-correct programs. If the mutation-based approach fails, a case analyzer detects missing program branches to guide students to partial solutions with reasonable structures.

After analyzing over 75,000 program submissions and 8,789 hint requests, we found that using all three techniques together could offer hints for any program, no matter how far it was from a correct solution. Furthermore, our analysis shows that hints contributed to students' progress while still encouraging the students to solve problems by themselves.

Keywords

Computer-Aided Education; Program Synthesis; Program Analysis; Automated Tutor

1. INTRODUCTION

In an introductory programming course, there are many ways for students to receive help, such as going to office hours to ask in person and posting to the online course forum. However, as course enrollment increases, it becomes harder to scale these support mechanisms. An automated approach offers a scalable alternative.

Many intelligent systems have been developed to automatically provide guidance to students completing programming exercises. Hint generation via path construction leverages existing student submissions to construct the most desirable path to a correct solution [1, 6, 4, 5, 7]. This approach can theoretically generate hints for any program, but in practice, it requires a tremendous amount of engineering effort for instructors to build a robust (high-coverage) hint generation system; the most robust system, ITAP, requires a large number of non-trivial program transformations to canonicalize students' programs and to undo the canonicalization [7].

Another popular approach, mutation-based, uses error models — or mutation rules, provided by the instructor — to mutate a student's incorrect program until it is semantically equivalent to a teacher's solution [8, 9]. Hints can then be naturally derived from the mutation rules that fix the program. While this approach requires far less engineering effort, it may fail to generate hints, especially when a student's program is not close to a correct solution.

This paper extends the mutation-base hint generation using complementary techniques to build a high-coverage hint generation system for Scheme assignments that:

- can provide hints for any program, no matter how far it is from a correct solution
- converts results from its internal algorithms to meaningful hints shown to students
- allows an instructor to add new problems and customize hint messages easily, and
- has been deployed and evaluated in a large introductory programming course with roughly 1,500 students and over 75,000 attempts on a single assignment.

2. OUR APPROACH

Our hint generation system was deployed in UC Berkeley's introductory computer science course. During the middle of the term, the course switched from using Python to Scheme. The course staff often complained that in Scheme office hours, they had to address the same question or similar syntax misconceptions repeatedly.

After reviewing student programs, we recognized three main categories of student errors. Instead of using a single approach to handle all kinds of errors, our system handles each kind differently.

The first category contains programs with semantic errors due to syntax misconceptions. Our observations revealed that this category covers a significant portion of incorrect programs because many students struggle with the placement of parenthesis in Scheme. For example, many students attempt to call a Scheme function with `f(x)` instead of the

ITiCSE '17, July 3–5, 2017, Bologna, Italy.

DOI: http://dx.doi.org/10.1145/3059009.3059058

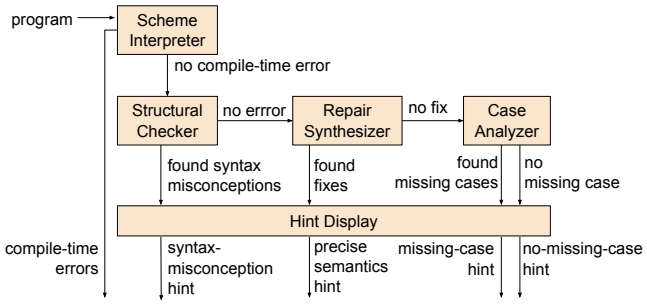

Figure 1: The workflow of the hint generator

```
The computer thinks that your program lacked or
had extra pairs of parentheses.
```

(a) High-level syntax misconception hint

```
>> Syntax expert:
Check the syntax of the conditional clause
at line 95.

Example(s) of correct syntax:
(cond ((> a b) (* a b)) (else (func a b)))

Example(s) of bad syntax:
(cond ((> a b) * a b) (else func a b))
```

(b) Detailed syntax misconception hint

Figure 2: Examples of Scheme syntax misconception hints

correct way, `(f x)`. To handle programs in this category, the system applies pattern matching on students' programs to check if there are common syntax misconception errors.

The second category contains programs that are almost correct. We handle this category by using the mutation-based hint generation approach. We improve upon the existing systems by introducing a more convenient way to encode error models for a new problem and allowing an instructor to customize a hint message associated with each error model.

The third category contains programs that are far from being correct. To handle programs in this category, we invent a case analysis technique to check if the student's solution contains all conditional checks (e.g. if/cond statements' conditions) appeared in the teacher's solution. Unlike the mutation-based technique, the case analysis technique does not know how to fix a student's program. Instead, it prompts the student to think about cases that he or she may have missed, thus, guiding the student toward a partial solution with a reasonable program structure.

Deploying all three different hint generation techniques together in a single system in a massive course showed that none of them alone would have been sufficient to generate helpful hints reliably.

3. HINT GENERATION

Figure 1 displays the workflow of our hint system, consisting of a Scheme interpreter and three hint generation components—the structural checker, the repair synthesizer (program mutator), and the case analyzer—corresponding to the three different types of programs described in Section 2. The system first checks a student's program with a Scheme interpreter to ensure there is no compile-time error before trying to generate:

1. a *syntax misconception hint* using the structural checker
2. a *precise semantics hint* using the repair synthesizer
3. a *missing-case hint* or *no-missing-case hint* using the case analyzer.

3.1 Structural Checker

The structural checker searches for an expression in a program that matches one of the invalid Scheme patterns described in Table 1. If the checker finds one or more invalid patterns, it displays a syntax misconception hint to the student. We designed the hint system to reveal details over time: it shows high-level hints within the first five attempts and then displays detailed hints after that. Most detailed syntax misconception hints not only point out mistakes but also provide examples of correct syntax for a similar expression to the student's problematic expression. Figures 2a and 2b show an example of a high-level and a detailed syntax misconception hint respectively. Note that the Scheme interpreter will not detect this type of error during compile time but will instead display a more obscure runtime error.

3.2 Repair Synthesizer

The repair synthesizer is based on the mutation-based hint generation system used by the EdX MITx 6.00x programming course [9]. We implemented a similar system for Scheme instead of Python. Given a student program and error models (mutation rules), the system applies the error models on the student program to generate all possible mutations of the student program. If the synthesizer finds a correct mutated program — semantically equivalent to the instructor provided solution — it generates a hint based on the mutations applied. Figure 3 displays an example of a hint generated from our repair synthesizer when it fixes a program using two mutation rules; the first rule deletes one of the conditions in `cond`, and the second rule adds a base case. Although the system knows how to fix students' programs precisely, it does not tell the students exactly how to do so. Instead, the system guides them to reach the correct solution by themselves.

We utilize Rosette [11], a solver-aided language, as a constraint solver to prove an equivalence of two programs. Rosette is particularly suitable for building a repair synthesizer for Scheme programs because Rosette is an embedded language in Racket of which Scheme is a subset. Unlike the prior work, which converts a student's Python program into the Sketch language [10], we do not need to convert programs into another representation to synthesize fixes.

Adding New Error Models

To enable the repair synthesizer, an instructor must provide students' error models. An error model captures a com-

```
The computer thinks that:
1. One of the conditions in 'cond' at line 9 is
   unnecessary and causes an error.
2. You may have forgotten to specifically handle
   some of these following cases
   or handle them incorrectly in function (f s):
     (number? s)
```

Figure 3: An example of a precise semantics hint

Construct	Error	Example
cond	missing a test expression or a body	(cond ((> a b) #t) (#f))
cond	missing a pair of parentheses around a body	(cond ((> a b) * a b) (else func a b))
cond	missing a pair of parentheses around a test expression	(cond (> a b #t) (else #f))
cond	missing a pair of parentheses around a test expression, body pair	(cond (> a b) #t else #f)
if	not matching (if test-expr then-expr else-expr)	(if (< a b) #t)
define	no body	(define (min a b))
define	multiple bodies that return non-void values	(define (min a b) (if (<= a b) a) (if (<= b a) b))

Table 1: A list of invalid Scheme patterns used in the structural checker

```
context: (cond ... (_ ?) ...)                [Rule A]
mutation: (cdr $x) => $x

context: (define (f $arg) ?)                 [Rule B]
mutation: $x => (cond ((= $arg 0) 1) (else $x))
hint: You may have forgotten to handle a base case
      when the argument is equal to 0.
```

Figure 4: Examples of mutation rules

mon mistake that students make, along with potential fixes. Some error models are applicable to most problems, such as an off-by-one error, using $\leq$ instead of $\geq$, and using *true* instead of *false*. Some error models are unique to a problem, such as missing base cases.

An improvement of our repair synthesizer over the prior work is a more convenient method to encode error models. In the prior work, instructors specify error models by overriding functions to mutate different types of AST nodes in students' programs. This method requires instructors to be familiar with the system's internals. Specifically, they must know about the mutation functions they need to override and the provided utility functions that can be used inside the mutation functions. A typical implementation of mutation functions for one question requires 300 lines of code.

In our system, instructors can conveniently encode error models by defining mutation rules without any knowledge of the system's internals. A rule consists of two parts: a context where a mutation can be applied and the mutation itself. Figure 4 shows examples of mutation rules.[1] The symbol `?` identifies where in the context the mutation should be applied. For Rule A, the context indicates that the mutation can be applied only to a body expression inside `cond`; `?` in `(_ ?)` indicates a body position; whereas `_`, which is a test expression, is ignored. Its mutation rule indicates that if a body matches `(cdr $x)`, we can try to replace the body with just `$x`; the symbol `$` informs the system that the term can match any expression. Rule B mutates a function f by adding a base case to return 1 if the argument to the function is 0. This rule is defined with a hint message, so if the repair synthesizer uses this rule to fix a solution, it will display this customized message.

3.3 Case Analyzer

When the repair synthesizer fails to provide hints, the program is passed on to the case analyzer, which reports the missing checks in the program with respect to all *conditional checks* extracted from the instructor's solution. Of course, there are multiple ways to implement a correct solution, and they may not use the same checks. However, we believe that if students are stuck, it may still be beneficial for them to think about scenarios that their programs have not handled.

To test if a conditional check from the instructor's program I, appears in a student's program S, we first collect all conditional checks in both I and S. We define a *conditional check* to be the test expression (e.g. if statement's condition) along with the path condition to the check. Consider programs in Figure 5: I in Figure 5a and S in Figure 5c contain the conditional checks shown in Figures 5b and 5d respectively. Notice that path conditions are included in the conditional checks. For example, the conditional check `i2` in Figure 5b is a conjunction of the path condition `(not (null? (cdr x)))` and the test expression `(<= (car x) (cadr x))` on line 4 of Figure 5a.

Then, we check if S has all the conditional checks that appear in I. Similar to the repair synthesizer, the case analyzer tests the equivalence of two conditional checks using Rosette. For this particular example, none of the conditional checks in I appear in S. Notice that although the check expression `(null? (cdr x))` appears in both programs, their path conditions to the check are not the same. We consider a path condition as part of a conditional check because it captures the order of conditional checks, which matters for the correctness of a program. In our running example, we must make sure that `(cdr x)` is not empty before we call `(cadr x))`; hence, `(null? (cdr x))` must be checked first.

```
1 (define (I x)
2   (cond
3     ((null? (cdr x)) #t)
4     ((<= (car x) (cadr x)) (I (cdr x)))
5     (else #f)))
```

(a) Instructor's program

```
check i1: (null? (cdr x))
check i2: (and (not (null? (cdr x)))
               (<= (car x) (cadr x)))
check i3: (and (not (null? (cdr x)))
               (not (<= (car x) (cadr x))))
```

(b) Conditional checks in the instructor's program

```
1 (define (S x)
2   (cond
3     ((< (car x) (cadr x)) (S (cdr x)))
4     ((null? (cdr x)) #t)))
```

(c) Student's program

```
check s1: (< (car x) (car (cdr x)))
check s2: (and (not (< (car x) (cadr x)))
               (null? (cdr x)))
```

(d) Conditional checks in the student's program

```
In your function (S x), what will happen if
the inputs to the (recursive) function meet
one of the following conditions? Does your
function handle these scenarios correctly?
  (null? (cdr x))
  (<= (car x) (cadr x))
  (and (not (null? (cdr x)))
       (not (<= (car x) (cadr x))))
```

(e) A missing-case hint generated for the student's program

Figure 5: How the case analyzer generates a hint

[1]The syntax of mutation rules used in this paper has been modified from the actual syntax used in our working hint generation system for the purpose of explaining the concept.

```
The computer believes that your program has
already covered all possible scenarios (different
conditions on the inputs), but the logic to handle
those scenarios are still incorrect.
```

Figure 6: No-missing-case generic hint

Furthermore, including path conditions reduces false alarms (reporting missing checks when there is no missing check) on programs with nested conditional statements and loops.

Once we have gathered all the missing conditional checks, we generate a hint accordingly. Figure 5e is the hint produced for the program in Figure 5c. We exclude most path conditions from a hint message to avoid complicating the hint. However, if the check expression alone is `#t` (such as `else`), we display the path condition. In the actual deployment, we set the system to print at most two missing cases in each hint so that we do not give away too much information.

In the scenario that there is no missing case, the system will print out a generic hint displayed in Figure 6.

4. SYSTEM DEPLOYMENT

We piloted our hint generation system for a Scheme assignment in UC Berkeley's introductory computer science course[2] in Spring, Summer, and Fall 2016. Students are graded on effort and completion in this assignment.

We integrated the hint system into the course autograder, called OK[3]. Students use OK through the command line to test their programs against instructor-provided tests. OK logs every time a student runs the autograder and sends the current copy of the program to the server [2]. To request a hint, students simply append `--hint` to the command for running the autograder. The system usually takes 1–10 seconds to generate a hint.

We used the Knowledge Integration framework to design the presentation of hints to the student [3]. Before providing hints, the system prompted the students to think about a particular comment from a list of instructor-selected prompts. The prompt encouraged students to reflect on their solutions before receiving new information via hints. Once the students had completed the problem, the system asked them to reflect on how the hint(s) changed their understanding of the solution.

The first deployment of the system in Spring 2016 revealed two major flaws of the system. First, the system was not able to provide hints for more than half of the requests due to the lack of the case analyzer. To resolve this, we developed the case analyzer. Second, we removed the requirement to respond to the Knowledge Integration pre-hint prompts so that we minimize the disruption to students. We deployed the improved system again in Summer and Fall 2016.

5. EVALUATION

We evaluated the effectiveness of the hint system by analyzing the data collected by OK in Fall 2016. In particular, we would like to answer two major questions:

1. Did hints help students complete the assignment?
2. Did students rely on hints in a way that may compromise learning?

[2]http://cs61a.org
[3]https://okpy.org

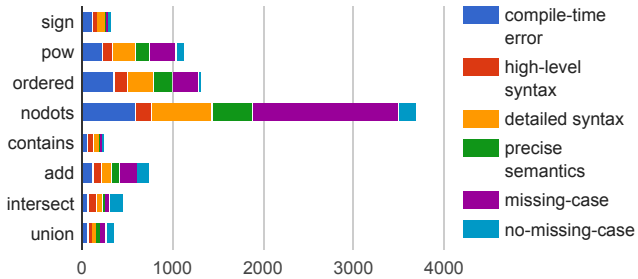

Figure 7: Number of hints per type per problem

The first question helps us determine if the system was helpful, while the second question determines if the system discouraged students from learning to solve problems by themselves. First, we start by presenting the general statistics on the hint usage in Section 5.1. Then, we answer the two central questions in Sections 5.2 and 5.3 respectively.

5.1 Hint Usage

A total of 1,485 students attempted the Scheme homework. The system logged approximately 75,000 student attempts as well as 8,789 hint requests.

918 students used the hint generation system at least once. The ratio of students asking for hints over all students range from 6.7%–41% across questions. Figure 7 displays the numbers of hints separated by question and types of hints. The order of the questions in the chart reflects the order presented to the students in the homework. As witnessed in the chart, `nodots` is the most difficult question and had the most number of hints requested. Also, notice that there are large portions of compile-time errors because most students in this course used the autograder not only to submit their programs but also to test and debug their programs. Apart from compile-time errors, 35% of hints were syntax misconception hints.

Among semantic hints, a majority of them were missing-case hints, which comprised 59% of all semantic hints, followed by precise semantic hints at 23%, and no-missing-case hints made up the remaining 18%.

The ability to generate precise semantic hints (23%) is far lower than that of the original work (64%) [9]. We hypothesize that the repair synthesizer performed worse in our real-world deployment because according to our survey, students requested hints mainly when they were stuck with solutions that were not close to being correct; as a result, these programs were harder to fix with a repair synthesizer.

5.2 Contribution Effect

To evaluate the overall contribution of the hint system, we compared the number of attempts students made on each problem in the assignment between Fall 2016 (with a hint system) and Fall 2015 (no hint system). An attempt is defined as an instance of a student locally running the autograder tests to determine if the current program is correct. Between the two offerings, the assignment as well as the instructor were identical, and the student population had similar demographics.

We found an 18.8% drop in the number of attempts made by students to get to a correct solution in Fall 2016, compared to that of Fall 2015. This reduction of the number of attempts was statistically significant ($p < 0.001$). The

effect was particularly pronounced for students in the upper quartile of the number of attempts, demonstrating that the hint system helped students make progress.

In the rest of this section, we evaluate the contribution effects of the different categories of hints separately, using the data collected in Fall 2016.

5.2.1 Syntax Misconception Hints

Many students struggled with syntax misconceptions, making many attempts while trying to resolve their syntax errors. Students who used our hint system for syntax errors had been struggling with the same error for an average of 4.69 attempts before requesting a hint. After receiving a hint, the median amount of attempts to change the error was one attempt. This metric shows that while our hint system did not result in students immediately resolving all of their syntax errors, it helped students move past their current error and advance towards a correct submission.

5.2.2 Semantics Hints

To evaluate the semantics hints, we manually inspect students' programs from the log files collected by OK. A single log file contained a sequence of program snapshots of a student solving a single problem over time along with the hints seen by the student. Of the 1,218 log files that contained at least one semantics hint, we randomly selected 89 log files to analyze, giving us a 95% confidence level with 10% confidence interval. For each log file, we evaluated the *reaction effect* and the *contribution effect* of the hints.

The reaction effect measures how frequently the student modified the program according to hints whether or not the changes were toward the right direction. The reaction effect gives us an idea of how useful the hints were at the moment. It is computed by dividing the number of times the student reacted to hints by the total number of the hints received.

A contribution effect measures how much the hints contributed to the student's final solution. Its value ranges from 1 to 5: 1) no contribution, 2) neutral (unsure), 3) little contribution, 4) moderate contribution, or 5) significant contribution to the final solution of the student. Note that a contribution effect of 5 does not imply that the hints gave away the answer, rather that the hints influenced the student to arrive at the correct answer.

The two authors scored the reaction effect and the contribution effect of the hints of each log file. The average scores from the two authors were then used in this analysis.

We analyzed the reaction and contribution effects obtained from two groups of log files separately. The first group A (452 log files) received at least one precise semantics hint (and possibly other types of semantics hints). The second group B (766 log files) received only missing-case and/or non-missing-case hints. We separated the two groups because hints from the case analyzer (missing-case and non-missing-case hints) are less informative than hints from the repair synthesizer (precise semantics hints). Note that the students in both of the groups might have also received syntax misconception hints, but we ignore the effects from the syntax misconception hints in this section.

Figures 8 and 9 display histograms of the reaction and contribution effects on group A and B, respectively. According to the reaction effects histograms, most students in both groups made changes related to the hints. However, there is a significant number of students from group B (24%) who

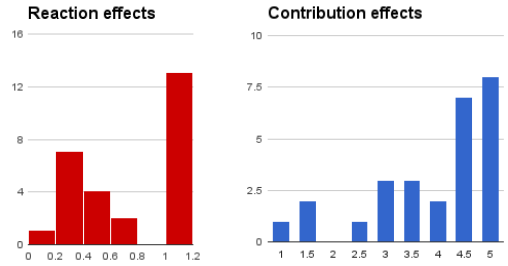

Figure 8: Histograms of effects from hints when students were receiving at least one precise semantics hint (group A). A reaction effect measures how often a student reacted to hints. A contribution effect measures how much hints contributed to a student's final solution.

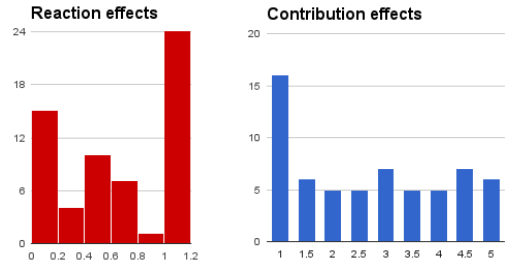

Figure 9: Histograms of effects from hints when students were receiving only missing-case and/or non-missing-case hints (group B)

rarely made changes related to hints (the first bin). Regarding the contribution effects, group A benefited from using hints 85% of the times, while only 48% for group B (bins 3–5). This result is not surprising because of two main reasons. First, hints from the case analyzer were more generic than hints from the repair synthesizer. Unlike the repair synthesizer, the case analyzer did not describe a specific fix for an incorrect program. Second, the case analyzer was only being evaluated on programs where the contribution of the mutation-based technique would have been zero because the mutation-based technique failed to generate a hint. Although the case analyzer did not appear to be as helpful as the repair synthesizer at first, it did help at least half of the students that the repair synthesizer could not help at all.

5.3 Dependence on Hints

The main concern with hints is that students may rely on hints too heavily and therefore avoid learning to solve problems by themselves. In this section, we show that students did not build dependence on the hint generation system.

5.3.1 Syntax Misconception Hints

Our hint system provided syntax misconception hints in the high-level form if a student made fewer than five attempts; otherwise, it displayed detailed information. We computed the ratio of high-level and detailed syntax misconception hints over all hints presented to students. Figure 10 shows that students who used the hint system received fewer detailed syntax hints over time, thus, more capable of fixing syntax misconceptions themselves.

5.3.2 Semantics Hints

We evaluated the rate at which students relied heavily on hints by analyzing a sample of the log files. Examining the

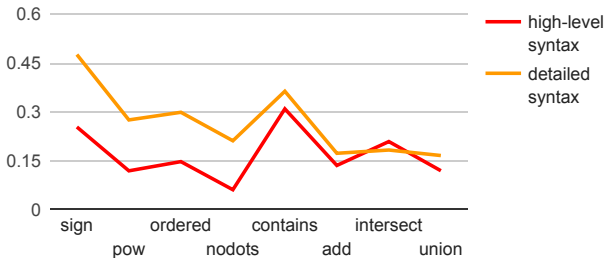

Figure 10: Ratios of high-level and detailed syntax misconception hints over all hints. The X axis is ordered from first to last problems in the assignment.

log files revealed that students who seemed to be mining the system for hints instead of trying to solve problems on their own obtained a large number of semantics hints (eight or more). There were a few students who did not rely on the hints heavily but received more than eight semantics hints. However, we took a conservative approach and categorized receiving eight or more semantic hints as relying too heavily on the system in a way that could compromise learning.

Out of students who received hints, the number of students who heavily relied on the system ranges from 0–3.8% across all problems except for `nodots`, which was 14%. Since `nodots` required many more attempts on average, it was more likely that these students actually needed more help and were not mining for more hints. Ignoring the outlier `nodots`, we conclude that only a small percentage of students misused the hint system and that the majority of students used the hint system as an assistant to help them learn.

6. DISCUSSION

The typical approach to individually helping students on programming assignments is difficult to scale to large courses. In this paper, we presented an automated hint generation system as a way to scale help to all students. We built on prior work to create a reliable hint system and deployed it in a massive CS1 course. The data collected from the deployment showed that applying different hint generation techniques enabled our system to provide hints reliably. Additionally, our hint system often helped students solving problems without having an adverse impact on learning.

According to the optional survey given to the students, there was a mix of both positive and negative responses. Many students expressed that the hint generation system was helpful: "it made the homework go faster, so I didn't have to wait for office hours or a response on [the online Q&A forum]." However, some students thought it was not: "it told me something I was already taking into account." Interestingly, some students expressed that they did not want any hint because "debugging the code on my own encourages more critical thinking and would help me learn more" and that hints would "diminish group collaboration."

Our experience suggests several ways that the system could be improved further. First, according to the survey, students expressed that some hints were not helpful because they contained Scheme expressions, which the students did not understand (e.g. hints in Figures 3 and 5e). To address this, we have customized hint messages in the repair synthesizer, and modified the case analyzer to output the missing cases in natural language and to suggest Scheme primitive functions that may be useful.

Secondly, instead of comparing a student's program to the instructor's solution (which may be very different), the case analyzer could compare the student's program to the most similar correct submission from all students. This way, the case analyzer can guide students toward more appropriate program structures for their current approaches.

Third, apart from providing more details for syntax misconception hints after five attempts, the hint system produces the same hint given the same student's program. Consequently, if students could not make any progress and ask for hints several times, they will receive exactly the same hint. Ideally, the system should detect this scenario and provide new information if possible or suggest students to review the material. As a step towards this, we have modified the system to provide a link to a Scheme tutorial when a student receives more than three syntax misconception hints on the same problem.

7. ACKNOWLEDGMENTS

We would like to thank Marcia Linn, Michael Clancy, and Eliane Wiese for their feedback on our system design; John DeNero and Andy Ko for their feedback on the paper.

8. REFERENCES

[1] T. Barnes and J. Stamper. *Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data*, pages 373–382. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[2] S. Basu, A. Wu, B. Hou, and J. DeNero. Problems before solutions: Automated problem clarification at scale. In *L@S*, 2015.

[3] M. C. Linn, H.-S. Lee, R. Tinker, F. Husic, and J. L. Chiu. Teaching and assessing knowledge integration in science. *Science*, 313(5790):1049–1050, 2006.

[4] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously generating hints by inferring problem solving policies. In *L@S*, 2015.

[5] T. W. Price, Y. Dong, and T. Barnes. Generating data-driven hints for open-ended programming. In *EDM*, 2016.

[6] K. Rivers and K. R. Koedinger. *Automating Hint Generation with Solution Space Path Construction*, pages 329–339. Springer International Publishing, Cham, 2014.

[7] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[8] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *ICSE*, 2017.

[9] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.

[10] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[11] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.