Summary of Progress and Future Plans for the Novel Logical Reasoning Tutor

Matthew F. Pfister,  
School of Computing  
Clemson University, Clemson, South Carolina 29631  
mpfiste@clemson.edu

Abstract—Students gain a deeper, more conceptual understanding of programming when they reason about code as opposed to running the code and observing output. Our research hopes to provide an easily approachable web-application that teaches basic programming concepts through reasoning – this is done through software verification and is independent of instructor aid.

I. OVERVIEW OF COMPLETED WORK

Thus far, we have completed the first-pass of an online logical reasoning tutor that teaches students the ability to reason about code through a series of ten challenges. The challenges increase in difficulty, covering a variety of basic programming constructs: assignments, if-then branches, and while loops. Students’ answers are checked for correctness using the RESOLVE verification system.

The tool has been tested by 100 sophomore level computer science students, and a second version has been in development based on user feedback. A summary of the feedback is provided as ratios of positive responses and negative responses for each of the survey questions, including exemplary quotes taken directly from the feedback:

<table>
<thead>
<tr>
<th>Feedback Topic</th>
<th>Positive Responses</th>
<th>Negative Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aesthetic</td>
<td>0.85</td>
<td>0.15</td>
</tr>
<tr>
<td>Interface</td>
<td>0.57</td>
<td>0.43</td>
</tr>
<tr>
<td>Language</td>
<td>0.74</td>
<td>0.26</td>
</tr>
<tr>
<td>Learning Experience</td>
<td>0.75</td>
<td>0.25</td>
</tr>
</tbody>
</table>

“I will never look at while loops the same again. But actually though, I mean that.”

“The website was a refreshing reminder on how code flows through conditionals and expressions.”

“It was an interesting perspective on what has to be true of variables at the start and end of functions.”

“I feel as though I have learned to look at code more closely, and to think more critically about what the code is actually doing.”

II. RECONSIDERATION OF SUMMARY METRIC

In testing the application, we recorded the number of attempts each student took to arrive at the correct answer for each of the ten challenges. We assumed the number of attempts taken would correlate to the difficulty the student had in completing the challenge – more attempts indicating more difficulty. If the number of attempts decreased from challenge to challenge, we thought we could safely conclude that the student was learning how to reason about the code.

However, in a recent seminar, Steven Edwards commented that the metric of attempt count is not singularly nor directly correlated to learning. Many confounding variables such as stress, focus, and confidence can influence the number of attempts taken on each of the challenges, rendering attempt count useless for drawing conclusions about learning, at least alone.

The remedy for this is to measure more than just attempt count, utilizing multiple metrics that together serve as a much more descriptive indicator to learning. With these additional metrics, we can develop a multi-parameter function that evaluates learning based on a weighted consideration for each of the parameters.

Once we have formulated this function, we anticipate being able to draw reliable conclusions about the application’s ability to teach students how to reason about code. However, the results produced by the function will not reveal whether this application is a superior tool for teaching compared to alternatives. That will require more thorough testing.

III. FUTURE RESEARCH GOAL

For this research to be of value to educators, we must prove a simply written yet weighty statement: teaching students using a conceptual approach is more effective than using the traditional approach. Here, the traditional approach is defined as trial-and-error based, focusing on the observation of the inputs and outputs of executing code to understand the code’s functionality.

This statement may seem trivially true, but not all educators are convinced, nor should they be. An unambiguous and rigorous test must be devised to demonstrate that one approach is definitively favorable over the other. Additionally, the test must clearly define what is meant by “more effective.”

We propose an experimental design involving a population of novice computer science students learning basic data structures – novice will likely mean sophomore level computer science undergraduates. The students will be randomly assigned to one of two experimental groups which solely vary teaching approach, either the traditional or the conceptual.

After completing instruction on basic data structures, the population of students will then be randomly given one of two exams testing their knowledge of these data structures. The exams will cover the same content, but one will be
contextualized by the traditional approach and the other by the conceptual approach.

We hope to see if there is a statistically significant difference in the students’ performance on these exams that can be contributed to teaching approach. Ideally, we will find that one approach causes significant gains in performance on both types of exam; if this is the case, we can confidently conclude that that approach is more effective.

IV. PROPOSITION OF SYMBOLIC EQUALS

In verifying the correctness of student provided answers to logical reasoning tutor, there have been many discussions about whether or not the context of the challenge should be considered. Whether we should consider context is in and of itself context sensitive, so we have provided examples of the more prominent cases.

Context-dependent Correct Answer

In the following code fragment, the text in bold is student provided input:

Challenge: Which variable is equal to the value held in Z?
```
Var X, Y, Z : Integer;
Read(Z);
Y := Z;
X := Z;
Confirm X = Z;
```

Now, assume that our expected answer to this challenge was Y, which is equally as correct as the student provided answer of X. In order to ensure our application accepts variant answers like X but still rejects incorrect ones, the tutor automatically appends an additional confirm to the code fragment before verification, one that compares the student answer to our own:

```
Confirm X = Y;
```

Since we are comparing expressions, we want to ensure that the expressions are equal to one another. In context to the rest of the code, this confirmation is true, and we accept the student’s answer.

Context-dependent Incorrect Answer

Consider the same example, now with the student providing an incorrect answer:

```
Challenge: Which variable is equal to the value held in Z?
Var X, Y, Z : Integer;
Read(Z);
Y := Z;
X := Z;
Confirm Z = Z;
```

To clarify, “incorrect” includes answers that are trivially correct, such as Z = Z. To do the same comparison of the student’s answer versus the key, the tutor appends:

```
Confirm Z = Y;
```

This confirmation, in context with the rest of the code, is verifiable, and the tutor evaluates the student’s input as correct. To address this problem, we propose a new key phrase to the language, Symbolic Confirm, which evaluates the truth of the statement without regard to the context of the problem. Here are some example evaluations:

```
Symbolic Confirm X = Y; -- false
Symbolic Confirm Y + 1 = 1 + Y; -- true
Symbolic Confirm X + 1 = 1 + Y; -- false
```

Only context independent transformations to the assertion are allowed with this new, symbolic confirm.

Now the tutor appends two confirmations after the student’s answer; the first ensures that the student’s answer is equivalent to the key, and the second ensures that the student’s answer is non-trivial, i.e cannot be confirmed by simple symbolic comparison:

```
Confirm Z = Y;
Symbolic Confirm not(Z = Z);
```

Since Z is so obviously equivalent to Z, the Symbolic Confirm evaluates the negation of the statement to be false, indicating that the student’s solution was trivial and thus incorrect.

Context-independent Incorrect Answer

Now consider a similar problem except this time we ask the student to provide a full conditional statement and not just an expression:

```
Challenge: What is the relationship between X and Y?
Var X, Y : Integer;
Read(Y);
X := Y;
Confirm true;
```

So in this challenge, the student might be compelled to confirm true rather than our expected answer of X = Y. Since we are comparing conditionals, we create an implication when comparing the student answer to our own, appending to the code:

```
Confirm true → (X = Y);
```

But since the basic Confirm construct considers the entire context of the program, this confirmation evaluates to true since X does in fact equal Y. This will always happen when considering the entire context of the code since our expected answers should be true in context, regardless of the student’s answer.
Here, we want to evaluate the student’s conditional out of context to the rest of the program:
Symbolic Confirm true → (X = Y);

This again reduces down to a statement which is clearly unverifiable outside of context:
Symbolic Confirm X = Y;

V. ACKNOWLEDGEMENT

This research is funded in part by NSF grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

And thank you to Murali Sitaraman for assisting in the writing of this paper and the development of the code examples.