Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics

Amruth N. Kumar
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430, USA
1 201 684 7712
amruth@ramapo.edu

ABSTRACT
An earlier study had found that solving code tracing problems helped improve code writing skills of students. But, given the instruments used in the earlier study, the improvement in code writing pertained primarily to language syntax. A follow-up within-subjects controlled study was conducted to investigate whether solving code tracing problems could help improve code writing skills pertaining to the semantics of a program. In the study, students were asked to write code for a control and a test problem both before and after a problem-solving session on code tracing. Increase in the score from pre-quiz to post-quiz was treated as improvement in code writing attributable to code tracing. Repeated measures ANOVA was used to analyze the data collected over four semesters. A statistically significant improvement in code writing skills pertaining to program semantics was observed on the test problems, but not on control concepts in the control problems. The improvement in code-writing skills as they pertain to program semantics accrued to the students who scored 90% or more on code-tracing problems in this study. Finally, the transfer in learning from code-tracing activities to code-writing skills may be near as well as far.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer Science Education

General Terms
Measurement, Performance, Experimentation

Keywords

1. INTRODUCTION
Code tracing and code writing are two of the activities used for both learning and assessing programming (e.g., [6]). Code tracing problems (e.g., debugging a program, identifying the output of a program) are attractive in that they can be solved in shorter stints of time, using formats such as multiple-choice that are easier to grade. In this context, a question of interest to Computer Science researchers has been whether there is a correlation between students’ performance on code tracing problems and their ability to write code [13] – if such a correlation exists, assessment in introductory Computer Science courses can be simplified and even automated. One study [10] found a positive correlation between code reading and code writing tasks using SOLO (Structure of the Observed Learning Outcome) taxonomy. Another study [1] found low correlation between code tracing and code writing problems, and yet another found a positive correlation between the two [8], which was later replicated with a follow-up study [12]. Other researchers have reported inability to replicate results establishing a relationship between the two [11]. Based on Piaget’s theory of intellectual development, yet others have argued that code-tracing is a legitimate phase in a novice student’s development into a programmer [7].

A recent study [5] reported finding a significant improvement in code-writing skills attributable to code-tracing activity. However, due to the topic selected for the study and the design of the instruments used in the study, the improvement found in code writing pertained primarily to language syntax. A follow-up study was conducted to see if code-tracing activity would benefit code-writing skills as they pertain to program semantics rather than language syntax. In this paper, the protocol, data collection and analysis, and results of the study will be discussed. This study is of interest because learning program semantics is harder than learning language syntax; and enabling students to passively learn program semantics by solving code-tracing problems would provide another pedagogic tool to Computer Science educators.

2. METHODOLOGY
In this study, the influence of solving code-tracing problems on the ability to write for loops was evaluated. A within-subjects controlled study was conducted – the students were asked to write code for a control problem and a test problem both before (pre-quiz) and after (post-quiz) solving code-tracing problems on for loops. The test problem pertained to writing a for loop, whereas the control problem pertained to writing a conditional break statement, a concept that was not covered in the intervening code-tracing problems. If code-tracing helps improve code-writing skills, significant improvement in code-writing skills should be observed on the test problem, but not the control problem.

2.1 Protocol
The protocol consisted of the following stages administered to the students back-to-back, with no break in between:
Pre-Quiz on code writing: Students were asked to write code for two problems:

1. **Pre-quiz control problem:** “A loop repeatedly reads values into the variable number and prints them. Write the code that must be inserted between reading and printing so that the loop is exited if the value of number is greater than 100.”

2. **Pre-quiz test problem:** “Write a for loop to print all the multiples of 5 from 25 up to and including the value of the integer variable limit.”

Problem-Solving session on code tracing: Students were presented a series of problems. In each problem, students were presented a program that contained one or more for loops, and asked to identify the output of the program – Please see Figure 1 at the end of the paper for a screen shot of a problem-solving session in progress. The problems were designed to test the students’ knowledge of the semantics of for loops, and their ability to trace programs. The concepts covered by the code-tracing problems included: zero-iteration loops, single iteration loops, dependent and independent nested loops, dependent and independent back-to-back loops, loops with simple and compound statement for body, up-counting and down-counting loops, and loops wherein the loop counter is updated again within the loop body. None of these problems included any other type of control statement (if, while, do-while), break or continue statements.

Post-Quiz on code writing: Students were again asked to write code for two problems:

1. **Post-quiz control problem:** “A loop repeatedly reads values into the variable temperature and prints them. Write the code that must be inserted between reading and printing so that the loop is exited if the value of temperature is less than 0.”

2. **Post-quiz test problem:** “Write a for loop to print all the multiples of 7 from 21 up to and including the value of the integer variable finish.”

Notes that the post-quiz control problem is isomorphic to pre-quiz control problem, i.e., identical modulo variable names, literal constants and output strings. Similarly, post-quiz test problem is isomorphic to pre-quiz test problem, in order to ensure that the difficulty level of pre- and post-quiz problems are comparable, as required for sound pre-post evaluation design. Control problems covered two constructs that were never included in any intervening code-tracing problem: if and break statements.

Pre-test-practice-post-test protocol was used for the session [4]: all the students initially answered a pre-test consisting of 10 problems, one per concept. If a student answered a pre-test problem incorrectly, the student was presented feedback explaining the correct answer, and was presented additional problems on the concept during the adaptive practice and post-test that followed. If a student solved a problem correctly during pre-test, no additional practice or post-test problems were presented on the concept. In order to stay within the typical attention span of learners, the entire pre-test-practice-post-test protocol was limited to 30 minutes.

Grading control problems: The answers to pre- and post-quiz control problems were graded in terms of 7 components, themselves combined into three subgroups:

- **Shell**, consisting of 2 components: the reserved word if, and parentheses;
- **Condition expression**, consisting of 3 components: the condition variable, relational operator and sentinel value;
- **Break statement**, consisting of 2 components: the reserved word break and the semi-colon that follows it.

Grading test problems: The answers to pre- and post-quiz test problems were graded along 16 components, themselves combined into five subgroups:

- **Shell**, consisting of 3 components: the reserved word for, parentheses and the two semi-colons within the parentheses;
• Initialization expression, consisting of 3 components: the loop counter, assignment operator and the initialization value;
• Condition expression, consisting of 3 components: the loop counter, relational operator and the terminal value;
• Update expression, consisting of 3 components: the loop counter, assignment operator and the step value;
• Loop action, i.e., the output statement, consisting of 4 components: the output statement (e.g., cout in C++, System.out.println in Java), output operator (<< in C++)/parentheses (in Java), the expression being output, and the statement delimiter semi-colon.

Note that the 3 components of shell and the 3 components of Loop action other than the expression being output all pertain to language syntax. The remaining 10 components – initialization expression, condition expression, update expression and the expression being output in loop action pertain to program semantics.

In order to reduce the time and effort needed for grading, while also maintaining consistency of grading, a Java program was written to automatically grade all 16 components of each student program. The grading program compared each student program against reference solutions, and awarded grade on a 3-point scale for each component as follows: 0 if the component was not used (e.g., no for or no output statement), 1 if it was incorrectly used (e.g., for in the wrong case, incorrect output statement), and 2 if it was completely and correctly used. The grading program resolved all the syntax components, but resolved only unambiguously correct semantic components.

Thereafter, all the student solutions were manually graded to resolve components that the grading program could not automatically grade. This was necessary in all the cases where the student’s program diverged significantly from the reference solutions built into the grading program. Manual grading was primarily needed for the 10 components pertaining to program semantics. In order to ensure consistent grading, especially of partially correct code, the following principles were followed:

• Partial credit was calculated based on the fewest text changes needed to make a student program completely correct, and not based on the concurrence of its output with that of the correct program, e.g., suppose a student wrote the following program for pre-quiz problem:
  ```
  for( var = 25; var <= limit; var + 5 )
  cout << var;
  ```
  Even though this is an infinite loop, it was penalized only for the assignment operator in the update expression, which should have been += instead of +.

• When grading semantic components, syntax errors were ignored, e.g., consider the following student code for pre-quiz problem:
  ```
  for( var = 5; var <= limit / 5; var++ )
  cout << 5 var;
  ```
The student was given full credit (2) for the output expression, even though multiplication operator is missing between 5 and var.

• Extraneous code such as variable declarations before the loop, declaration of loop counter within the loop header and inclusion of endl in C++ output statement was ignored.

• Incorrect use of sentinel variable as loop counter was graded as incorrect at each occurrence, e.g., consider the following student code for post-quiz problem:
  ```
  for( finish=21; finish<=100; finish+=7 )
  System.out.println( finish );
  ```
  Since finish holds the sentinel value, its use was graded as incorrect (1) in the initialization expression, condition expression, as well as update expression.

Since the focus of this study was semantics, the total score on pre-quiz and post-quiz was calculated as the sum of the scores on the 10 semantic components – initialization, condition, update and output expressions, for a maximum score of 20 on both pre-quiz and post-quiz problem.

Scoring code-tracing problems: Students also received a score on the code-tracing problems they solved during the pre-test-practice-post-test problem-solving session. On each problem, they received partial credit for all the outputs they correctly identified in a program up to the first incorrect output. Given that practice and post-test were adaptive, pre-test was the only stage attempted by all the students. So, only pre-test stage data was used in this study. Since the problem-solving session was limited to 30 minutes, the number of pre-test problems solved varied among the students. So, for analysis purposes, the score per problem (normalized to a maximum of 1.0) and the time spent per problem (in seconds) were used instead of total score and total time.

3. RESULTS
First, pre-post change in score will be analyzed for test problems, in order to find out whether students could write better code after the problem-solving session on code tracing, and if so, in which component(s) pertaining to program semantics. Next, pre-post change in score will be analyzed for control problems to see if any improvement in the score on test problems may merely be due to Hawthorne effect [3]. Finally, quiz scores will be compared with the scores on the code-tracing session to see how students who knew code-tracing did on code-writing and vice versa.

3.1 Analysis of semantic components
Test problems: Repeated measures ANOVA analysis of the total score yielded significant main effect [F(1,311) = 7.093, p = 0.008], i.e., in a test with one degree of freedom between subjects (two groups) and 311 degrees of freedom within subjects (312 students), the ratio of the mean squares between-subjects and within-subjects, also called F-ratio, was 7.093. (The F-ratio is a measure of how different the means are relative to the variability within each group - the larger the F-ratio, the greater the likelihood that the difference between the groups is due to more than chance alone.) The significant main effect indicates that the treatment (code-tracing session) indeed had an effect on the pre-post change in scores on test problems. Finally, the result was statistically significant since the probability p was less than 0.05.

Post-hoc tests revealed that students scored higher on the post-quiz than on the pre-quiz as shown in Table 1, and the improvement was statistically significant (means are listed with
confidence intervals at 95% confidence level). Since pre-quiz, code-tracing session and post-quiz were administered back-to-back with no break in between, the pre-post improvement in quiz score can be attributed to the intervening problem-solving session on code tracing. However, the effect size was small (0.11), i.e., the test group mean was at 54 percentile of control group.

**Table 1. Total score on test problems before and after solving code-tracing problems**

<table>
<thead>
<tr>
<th>Score (out of 20)</th>
<th>Pre-Quiz</th>
<th>Post-Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>17.051 ± 0.419</td>
<td>17.455 ± 0.377</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>3.763</td>
<td>3.384</td>
</tr>
</tbody>
</table>

The maximum score on the test problem was 20. Those who scored 19 or 20 on the pre-quiz had little or no room for pre-post improvement in score. In order to eliminate this ceiling effect, analysis was re-run with only the students who had scored less than 19 points on the pre-quiz, i.e., 154 students. Once again, a significant main effect was found [F(1,153) = 12.43, p = 0.001]. The F-ratio corresponding to the change in pre-post scores was much larger at 12.43, and the result was once again statistically significant. The improvement in score from pre-quiz to post-quiz was much larger too, as shown in Table 2. The effect size was larger (0.25), considered small to medium.

**Table 2. Total score on test problems before and after solving code-tracing problems – after eliminating ceiling effect**

<table>
<thead>
<tr>
<th>Score (out of 20)</th>
<th>Pre-Quiz</th>
<th>Post-Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>14.571 ± 0.357</td>
<td>15.565 ± 0.395</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>4.037</td>
<td>3.799</td>
</tr>
</tbody>
</table>

Post-hoc analysis of the 10 semantic components was conducted to identify the components on which students had shown significant improvement in code writing skills. Once again, ceiling effect was accounted for by eliminating students who had scored 19 or 20 on the pre-quiz test problem. Significant pre-post improvement was found on the following components:

- Initialization operator: [F(1,153) = 6.43, p = 0.012]: the mean score improved from 1.76 to 1.896 (out of a maximum of 2 points)
- Initialization value: [F(1,153) = 19.304, p < 0.001]: the mean score improved from 1.247 to 1.474
- Update operator: [F(1,153) = 9.865, p = 0.002]: the mean score improved from 1.26 to 1.422
- Update value: [F(1,153) = 7.144, p = 0.008]: the mean score improved from 1.545 to 1.695.

So, the improvement in the ability to write for loops occurred in the students’ ability to start and step the loop counter. These four components pertain to program semantics rather than language syntax.

**Control problems:** Control problems covered two constructs that were not included in any code-tracing problem: if and break statements. If shell in the control problems pertains to syntax and break statement pertains to semantics.

Repeated measures ANOVA analysis found no significant main effect for break statement [F(1,67) = 0.33, p = 0.568]: among the 68 students who attempted to write break statement on pre- and post-quiz, no statistically significant improvement was observed from pre-quiz to post-quiz, as shown in Table 3. Since the maximum score on break statement was 2, the lack of improvement cannot be attributed to ceiling effect.

**Table 3. Score on break statement in control problems before and after solving code-tracing problems**

<table>
<thead>
<tr>
<th>Score (out of 2)</th>
<th>Pre-Quiz</th>
<th>Post-Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1.574 ± 0.153</td>
<td>1.588 ± 0.158</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>0.076</td>
<td>0.079</td>
</tr>
</tbody>
</table>

Similarly, repeated measures ANOVA analysis found no significant main effect for if statement [F(1,140) = 0.307, p = 0.58]: among the 141 students who attempted to write if statement on pre- and post-quiz, no statistically significant change was observed from pre-quiz to post-quiz, as shown in Table 4. Once again, since the maximum score on if statement was 4, the lack of improvement cannot be attributed to ceiling effect.

**Table 4. Score on if statement in control problems before and after solving code-tracing problems**

<table>
<thead>
<tr>
<th>Score (out of 4)</th>
<th>Pre-Quiz</th>
<th>Post-Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>1.972 ± 0.333</td>
<td>1.936 ± 0.332</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>0.168</td>
<td>0.168</td>
</tr>
</tbody>
</table>

So, solving code-tracing problems did not help students learn syntactic or semantic constructs that were not included in code-tracing problems. This discredits alternative explanations (e.g., Hawthorne effect – improved performance due to awareness of being observed) for the improvement observed earlier in code-writing due to code-tracing.

Condition expression of an if statement is similar in semantics to the condition expression of a for loop. This explains why ANOVA analysis of the condition expression yielded significant main effect [F(1,124) = 9.271, p = 0.003]: a small, but significant increase was observed in the mean score from pre-quiz (5.432 ± 0.134) to post-quiz (5.592 ± 0.115). Post-hoc analysis showed that the significant improvement was only in writing the conditional operator [F(1,124) = 7.089, p = 0.009]: the mean score increased from pre-quiz (1.6 ± 0.09) to post-quiz (1.704 ± 0.081), and the increase was significant. This indicates that the transfer in learning from code-tracing activities to code-writing skills may be near as well as far [9]: near transfer being defined as when original (tracing for loops) and transfer (writing for loops) contexts are similar, and far transfer being defined as when original (tracing for loop condition) and transfer (writing if statement condition) contexts are dissimilar. Further studies are needed to evaluate whether the example of far transfer qualifies as non-specific transfer [9].

### 3.2 Analysis of subjects

Next, the subjects were partitioned into two groups based on their score during the problem-solving session on code tracing: those who scored 90% or more versus those who scored less than 90% (referred to as sub-90%). Mixed factor ANOVA analysis was conducted with quiz scores on test problems as repeated measure...
and 90%+ versus sub-90% grouping as the between-subjects factor. No significant main effect was found for group [F(1,310) = 0.52, p = 0.471]. But, the quiz scores of both the groups increased from pre-quiz to post-quiz, as shown in Table 5. Ironically, the pre-post improvement was significant for 90%+ group [F(1,197) = 5.561, p = 0.019], but not for sub-90% group [F(1,113) = 1.533, p = 0.218]! So, the improvement in code-writing skills as they pertain to program semantics accrued to the students who scored 90% or more on code-tracing problems in this study.

### Table 5. Pre and Post-quiz test problem scores: 90% versus sub-90% scorers on problem-solving session

<table>
<thead>
<tr>
<th>Score (out of 20)</th>
<th>Pre-Quiz</th>
<th>Post-Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%+ scorers during problem-solving session (N=198)</td>
<td>Mean: 17.111 ± 0.527</td>
<td>17.601 ± 0.473</td>
</tr>
<tr>
<td>Std Deviation: 3.928</td>
<td>3.261</td>
<td></td>
</tr>
<tr>
<td>Sub-90% scorers during problem-solving session (N=114)</td>
<td>Mean: 16.947 ± 0.694</td>
<td>17.202 ± 0.623</td>
</tr>
<tr>
<td>Std Deviation: 3.471</td>
<td>3.588</td>
<td></td>
</tr>
</tbody>
</table>

Finally, the subjects were partitioned into three groups based on the change in their score on test problem from pre-quiz to post-quiz: negative, zero and positive. Negative change meant code on the post-quiz qualified for fewer points than code on the pre-quiz; positive change meant improvement in coding from pre-quiz to post-quiz.

The mean score per problem during the problem-solving session on code tracing was analyzed using univariate ANOVA, with the group being the fixed factor. No significant main effect was found for group [F(2,311) = 0.041, p = 0.96], i.e., the code-tracing skills of the three groups are not significantly different. Similarly, analysis of the mean time spent per code-tracing problem did not yield a significant main effect for group [F(2,311) = 0.804, p = 0.449].

### 4. DISCUSSION

It was surprising to find that the students whose code-writing skills significantly improved, as they pertain to program semantics, were the ones who scored 90% or more on code-tracing problems, presumably, the better-prepared students. One needs a good mental model of programming [2] in order to be able to trace code. We argue that the availability of such a mental model helps these students pick up code-writing skills, as some have recently argued (e.g., [7])! Or, can a novice learn to write code without learning to trace it first? These questions could benefit from additional studies of the relationship between code-tracing and code-writing.

One of the most challenging aspects of this study was manual grading of student code - it was found that even with the benefit of codified grading principles, it was quite easy to arrive at two entirely different grades for the same snippet of code, especially when the grader took into account speculations about the student’s design and thought process. Given this, there is likely to be some inconsistency in grading, especially when hundreds of students are involved in the study. It is hoped that the large sample size counters any adverse effects of stray inconsistencies in grading.

A three-point scale was used for grading in this study. A more elaborate grading policy might better elicit subtle differences between student populations, and correlations between code tracing and code writing skills. Then again, such a policy would entail larger commitment of resources for grading.

The hypothesis of this study was that tracing code leads to improvement in code-writing skills, not that tracing code is necessary for improving code-writing skills. Therefore, the within-subjects design of the evaluation protocol was adequate: while other activities such as reading about loops may also lead to improvement in code-writing skills, this study establishes that so does code-tracing activity.

The results of this study, combined with that of an earlier study [5], indicate that code-tracing activity helps students learn to write both syntactic and semantic components of code. However, the effect size was found to be small-to-medium in both studies. So, code-tracing can be used as a supplement to rather than a substitute for code-writing exercises.

Our serendipitous discovery of far-transfer of learning from code-tracing to code-writing is promising, and warrants additional studies. Could there be a few key programming concepts (e.g., condition expression evaluation, transfer of control in statements, side-effect tracking), tracing code on which might lead to disproportionate gains in code-writing skills in general? We plan to start by investigating whether evaluating expressions will help students learn to write the correct expression for a given problem.

### 5. ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation under grants DUE-0817187 and DUE-1432190.

### 6. REFERENCES


Figure 1 (© Amruth N. Kumar): Screen shot of a problem-solving session on code-tracing: a C++ program involving nested for loops is presented in the left panel. The student has entered the first two outputs of the program in the bottom right panel. Each output includes the printed value as well as the line number of the line of code that produces that output. Instructions for entering the answer are shown in the top right panel.