Trace-Based Teaching in Early Programming Courses

Matthew Hertz
Computer Science Department
Canisius College
Buffalo, NY 14208
hertzm@canisius.edu

Maria Jump
Department of Math and Computer Science
King's College
Wilkes-Barre, PA 18711
mariajump@kings.edu

ABSTRACT

Students in introductory programming courses struggle with building the mental models that correctly describe concepts such as variables, subroutine calls, and dynamic memory usage. This struggle leads to lowered student learning outcomes and, it has been argued, the high failure and dropout rates commonly seen in these courses. We will show that accurately modeling what is occurring in memory and requiring students to trace code using this model improves student performance and increases retention.

This paper presents the results of an experiment in which introductory programming courses were organized around code tracing. We present *program memory traces*, a new approach for tracing code that models what occurs in memory as a program executes. We use these traces to drive our lectures and to act as key pieces of our active learning activities. We report the results of student surveys showing that instructor tracing was rated as the most valuable piece of the course and students' overwhelming agreement on the importance of the tracing activities for their learning. Finally, we demonstrate that trace-based teaching led to statistically significant improvements student grades, decreased drop and failure rates, and an improvement in students' programming abilities.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computers and Information Science Education—*Computer science education; Curriculum*

General Terms

Tracing; Pedagogical Choices

Keywords

CS1, CS2, Pedagogy, Tracing

1. INTRODUCTION

Past research found that many introductory programming students hold impossible mental models of how programs execute. These inviable mental models include ones related to understanding how variables behave [7], how references work [14], and how subroutines get executed [3, 5]. It has also been suggested that the difficulty of developing viable models is among the leading reasons these early courses have such high dropout rates [11].

As a result, visual models were investigated as a way of helping introductory students. Nearly a decade ago, Holliday and Luginbuhl documented that a student's ability to draw visual representations of objects in a program's heap and their comprehension of the material were correlated [6]. This led Sorva and Sirkiä to determine that using a program which presents a program's objects visually helps students construct viable models of execution [15]. Given this potential to help students, many programs have been developed that perform this visualization (e.g., [1, 4, 15]).

This paper presents *program memory traces*, a new trace format that includes abstract representations of both a program's stack and its heap. By abstracting what happens in memory, our program memory trace can be used with many languages and for multiple programming paradigms.

We build on previous research by presenting our experiences organizing instruction in introductory computer science courses around tracing code using this new format. For this study, lectures typically begin with 20–30 minutes of instruction which followed brief introductions to the concepts with the instructor leading the class through tracing prewritten code samples that illustrate the concept. Students spend the remainder of the 50- or 75-minute lecture period working in small groups on activities focused on their tracing sample programs. By actively providing a visual model of the concepts and reinforcing this model with in-class activities in which students trace prewritten code, we help students develop viable models of the concepts and thereby improve student learning and increase the rate at which students successfully complete the course.

To demonstrate the effectiveness of our teaching approach, we examined both quantitative and qualitative measures of student learning and teaching effectiveness. We show that switching to this new approach increased the rate at which students successfully complete a Java-based data structures course from 75% to 91%. This was also met with students' unsolicited comments expressing their belief in the value of these traces. We also show that this increase was accompanied by a considerable improvement in students' lecture grade and a statistically significant increase in students' laboratory grade. Anonymous surveys from both a Java-based data structures course and a C-based introduction to programming course show that a majority of students in each course rated both watching the instructor trace code and tracing code on their own to be "very helpful" to their learning. Furthermore each form of tracing was rated either "helpful" or "very helpful" by at least 90% of each course's students.

The rest of the paper is organized as follows. Section 2 presents an overview of the program memory trace format we used. The organization of courses to focus on the use of these traces is described in Section 3. We analyze the effects of our trace-driven teaching approach in Section 4. Section 5 describes the related work and Section 6 offers suggestions for future work and concludes.

2. PROGRAM MEMORY TRACES

Motivated by a desire to help novice programmers develop valid mental models of many challenging programming concepts, we developed *program memory traces*. A *program memory trace* provides an accurate representation of the program's memory usage in such a way that it balances the abstractness required by introductory students with the accuracy needed for advanced students. We wanted to develop a model that would remain useful to students throughout their education and beyond. As such, a *program memory trace* depicts all of the program's accessible memory by dividing the trace area into different regions representing different types of memory: *stack*, *heap*, and *static*.

The first and most important region represents the program stack; we simply call it the *stack*. In the *stack*, we use blocks to represent the stackframes created whenever a subroutine is called. For each parameter and each local variable, the block contains a space for that variable including its name and its current value. Value variables represent their value in the space within the stackframe, while a reference variable's value is shown as an arrow pointing to the memory block to which they refer. Whenever a method returns (or quits executing), the corresponding block in the stack is disposed of by drawing an "X" through it. By accurately representing each method call with its stackframe, students are able to better understand method calling and variable scoping.

Next is the *heap* where blocks depict dynamically allocated memory. In C this includes any memory allocated using `malloc`, while in Java this includes memory allocated using `new`. In a program memory trace, the return values of `malloc` and `new` are depicted as an arrow from the reference variable to which the return type is assigned to the memory block representing the dynamically allocated memory. Blocks in the heap are disposed of by drawing an "X" through it. This would occur whenever they are freed (in C) or, optionally, are no longer reachable (in Java).

Finally, the *static* region contains any statically allocated memory. For static memory, blocks represent each statically allocated variable labeled according to who allocated it and thus who has the ability to modify it. Since static memory is allocated at the start of a program and "lives" until the end of the program, blocks in the static region are created when they are needed by the trace and are never disposed of.

To illustrate both how these program memory traces work and how they can be used with different languages and programming paradigms, we show two sample programs shown in Figure 1. This figure shows the same program written in both C (Figure 1a) and Java (Figure 1c) and the program memory traces generated for them (Figures 1b and 1d, respectively). These sample programs are intended to illustrate the trace format and so were designed to be small and highlight multiple features of the trace; they are not programs we have used in our classes.

In a *program memory trace*, program execution starts at a method call. In many cases, this is `main` but it does not have to be. At the start of any method, a new stackframe is created and, thus in our trace, we draw a block on the stack to represent that method. The method's parameters are evaluated and given space in the method's block. A common mistake when evaluating the method's parameters in an object-oriented language like Java is to forget the `this` reference. `this` is treated like a method parameter. Once the parameters are evaluated and their initial values recorded, the method can be stepped through one line at a time. With each line, we update the memory trace to reflect its effects. As an example, an assignment statement requires evaluating the right-hand-side. The right-hand-side could simply be a value, an arithmetic expression, or even another method call. The result of the evaluation is used to update the value of the variable named on the left-hand-side of the assignment. These updates are shown by crossing out the old value of the variable and showing the new value. When a method call is encountered, the execution of the current method is interrupted and a new stackframe created at the top of the stack. For most languages, only the method of the most current stackframe can be traced.

3. TRACE-DRIVEN TEACHING

Like many institutions [11], our introductory classes suffered from low retention rates and poor student performance. These problems occurred despite our making a number of pedagogical changes including adding active learning exercises and demonstrating how code features worked using live coding and canned examples. Students seemed to have an especially difficult time with issues such as variable scoping and pointer assignments/aliasing. These problems led to further difficulties. Despite devoting a week or more of class time to the basics of recursion, the majority of students struggled to write recursive methods. These classes slogged on with little of the fun and none of the "passion, awe, and joy" that encourage students to become or continue as computer science majors. Something needed to be done.

While many others had previously documented these *struggles* (e.g., [3,5,7,14]), very few solutions were suggested. The classes already included in-class code tracing and coding activities, labs, small programming homework exercises, larger programming projects, and other opportunities for students to develop valid mental models. We observed that some of these activities seemed to hurt as much as they helped. Even working in groups, activities in which students coded on paper seemed to reinforce inaccurate models and made it more difficult for students to change. Code tracing activities exposed students' inviable models [6] and, more importantly, provided a much clearer starting point with which to discuss correct models. While this should have encouraged students to trace code, students would avoid these questions and instead just guess at program's end state. As others have found, students were *extremely* hesitant to perform any tracing on their own [9, 16].

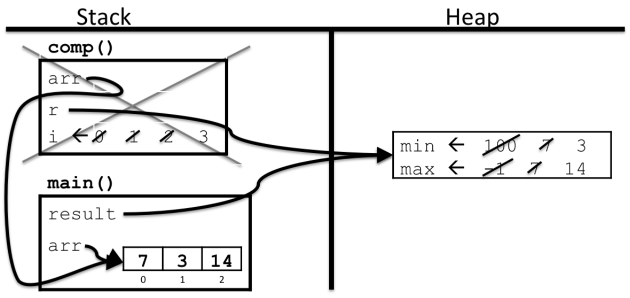
To improve upon our courses, we decided to surround students with viable models of memory by using traces as the focus of our teaching. We introduce students to tracing and the ideas of a *program memory trace* from the onset of our classes. While concepts such as the heap and stackframes are not needed immediately, we use them in our traces from the beginning to accurately model what is happening in memory. In our lectures, we follow each topic introduction with tracing through sample code to illustrate what the concept does or how it works. Because we continue to use active learning pedagogy, at least half each lecture period has students working on activities in small groups. These activities **always** include problems requiring they trace code and, if they ask students to write code, require students to trace the code they wrote [1].

Our hope was that by organizing instruction around *program memory traces*, students would be more likely to develop viable mental models of how programs execute. Beyond just surrounding them with visual representations, trace-based teaching should help

[1]Examples of these activities can be found at `http://cs.canisius.edu/~hertzm/tracing` and `http://staff.kings.edu/mariajump/tracing`

```
typedef struct Ex {
  int min, max;
} Ex;
void comp(int *arr, Ex *r) {
  for (int i=0; i < 3; i++) {
    r->max = (arr[i]>r->max)?arr[i]:r->max;
    r->min = (arr[i]<r->min)?arr[i]:r->min;
  }
}
int main() {
  Ex *result = malloc(sizeof(Ex));
  int arr[] = {7, 3, 14};
  result.min = 100;
  result.max = -1;
  comp(arr, result);
  return 0;
}
```

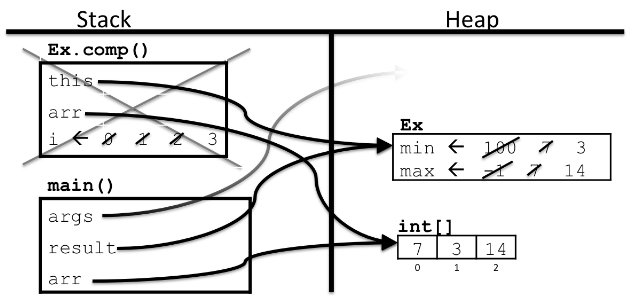
(a) Sample program written in C.



(b) Program memory trace generated for the C program.

```
public class Ex {
  private int min, max;
  void comp(int[] arr) {
    for (int i=0; i < 3; i++) {
      max = (arr[i]>max)?arr[i]:max;
      min = (arr[i]<min)?arr[i]:min;
    }
  }
  public static int main(String[] args) {
    Ex result = new Ex();
    int arr[] = {7, 3, 14};
    result.min = 100;
    result.max = -1;
    result.comp(arr);
    return 0;
  }
}
```

(c) Sample program written in Java.



(d) Program memory trace generated for the Java program.

Figure 1: Sample program, written in both C and Java, with which we illustrate how program memory traces are created. Both program use static and dynamic allocation, arrays and scalar variables, and subroutines. These will be used to illustrate how these traces work.

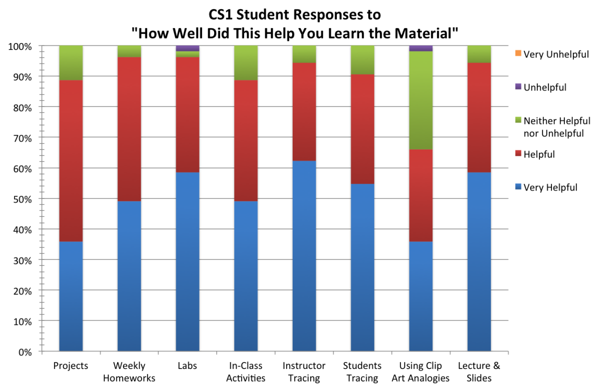
students by "priming" them for many of the concepts with which past research had found students struggle. One example of this is that of stackframes. By including stackframes from the very start, students are comfortable with the mechanics of variable scoping before the course gets to subroutine execution. Similarly, by having our traces represent what occurs in memory, students literally see the parallels between related concepts (e.g., arrays and pointers in C are both represented as arrows). These illustrations highlight these relationships and make reusing knowledge easier.

4. RESULTS AND ANALYSIS

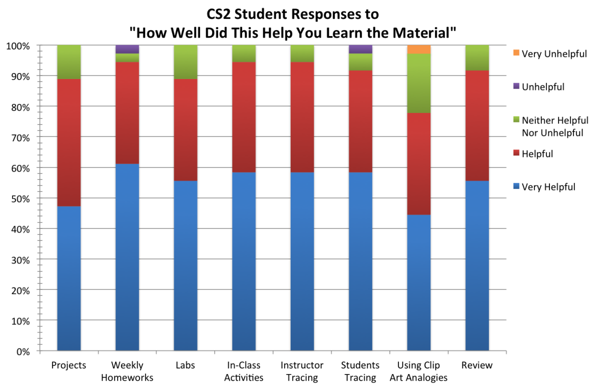
We use this trace-driven teaching approach in two introductory courses at a medium-sized regional college. The first, a CS1 ("Introduction to Programming") course taught in C, is required for, and largely populated by, first-year physics and pre-engineering majors. For nearly all of the students, this CS1 course is their first computer science course and they begin the term with little interest in computer science. Enrollments in this course vary greatly between 12 – 31 students. The second course, a CS2 course taught in Java, is required for computer science majors and minors and is usually taken by second-year students. Course registrations vary from year to year from a low enrollment of 9 to a high of 21 (mean enrollment was 14). A small minority of students come from the CS1 course described above, but would have had to have taken a Java-based CS1 course in the interim. Both the CS1 and CS2 classes in this study have a similar structure including 3 hours per week of instructional time, a 1 hour per week laboratory session, small weekly programming sets, and 3 larger programming projects. Classes are taught using active learning with 15 - 30 minutes of active lecture followed by activities performed in 3 - 5 person groups.

As a quantitative measure of the effectiveness of our teaching approach, we compare students' grades from the 3 years preceding the change to trace-driven teaching with the 4 subsequent years. This comparison can only be performed for the CS2 course because the content and coverage of the CS1 course changed significantly during this time frame.[2] As a qualitative measure, we performed end-of-course assessments during the last week of the course and, during the past 3 years, included questions asking students to rate how helpful each aspect of the course was for their learning the material. Responses to this question were on a 5-point Likert scale ("very unhelpful", "unhelpful", "neither helpful nor unhelpful", "helpful", and "very helpful"). Student participation in the end-of-course assessment was both voluntary and anonymous. Aggregated within each course, we had 53 responses in our CS1 course (a 88.33% response rate) and 36 responses in our CS2 course (a 97.30% response rate). As with most qualitative measures, students often confuse what was most helpful with what they liked the most. Regardless,

[2]In particular, the CS1 course now covers 25% more material and goes deeper into other topics relative to the course before switching trace-driven teaching



(a) Percentage of students answering how useful each of the aspects of the CS1 course was to their learning (n=53). Almost two-thirds of the students found the instructors tracing code in lectures to be "very helpful," a rate higher than any other aspect of the course.



(b) Percentage of students answering how useful each of the aspects of the CS2 course was to their learning (n=36). The instructor's use of traces continued to be more popular than even the lectures themselves and, along with weekly homework assignments, the aspect of the course that students found to be most helpful.

Figure 2: Results of anonymous surveys of the students in the CS1 (a) and CS2 (b) courses. Students in both classes found traces to really help them learn the material. In particular, students rated the instructor tracing code in the lecture as among the most helpful aspects of the course.

qualitative results often help explain any changes we see in the grades students received.

Students were overwhelmingly positive about using traces to explain the concepts. A majority of students in each class responded that the instructor's tracing code while lecturing was "very helpful" to their learning of the material. Over 94% of students in each class, in fact, stated that the instructor tracing code was either "helpful" or "very helpful" to their learning (the remaining students rated this as "neither helpful nor unhelpful"). As can be seen from the results in Figure 2a, students were more positive about the instructor's tracing of code than they were about the slides which included the traces. In fact, the instructor's tracing of code was found to be more helpful than the lectures as whole. Additionally, the data in Figure 2b shows that students rate the instructor tracing code among the most helpful aspects of both the CS1 and CS2 courses.

While students reported that the instructor tracing code helped them learn, our course design also places an emphasis on having the students trace code. While students had tried to "guess" what code does rather than actually complete a trace previously, students in trace-driven teaching courses willing trace code when a problem requires it .Even with this emphasis, students in trace-driven teaching class were no more likely to use traces on their own than students in our classes before the change. A majority of students in each class responded that they found having to trace code to be "very helpful". While the percentages are lower than for the instructor tracing code, over 90% of students in each class rated students tracing code positively (in fact, only a single CS2 student rated this "unhelpful"). In both courses student tracing code was rated more positively than the multi-week programming projects and in CS1 students tracing code was rated more positively than the daily activities as a whole.

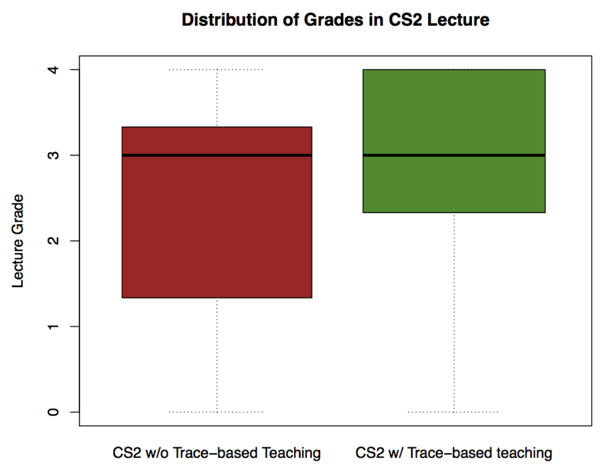
While students clearly feel that trace-driven teaching helps them learn, those results cannot show whether it improves student learning. For this we turn to quantitative data and compared students grades from the 3 years preceding the change to trace-driven teaching with the 4 subsequent years. All of these classes were taught by the same instructor using the same structure, content, and grading scheme. Any changes, therefore, reflect the effect of using trace-driven teaching.[3]

First we examine student's CS2 lecture grade which includes all aspects of the course except weekly labs. The CS2 lab (like the CS1 lab) require students work individually at a computer to complete the programming problems. These labs last 1 to 2 hours and assignments are due at the end of that time. The lab grade reflects students' performance writing the required code (the lab assignments do not require tracing, though students can use this to find bugs). As shown in Figure 3a, the median CS2 lecture grade remains unchanged but the distribution of grades has skewed much higher. Similarly, the CS2 lab grade distribution (see Figure 3b) show grades improved considerably since switching to trace-driven teaching.

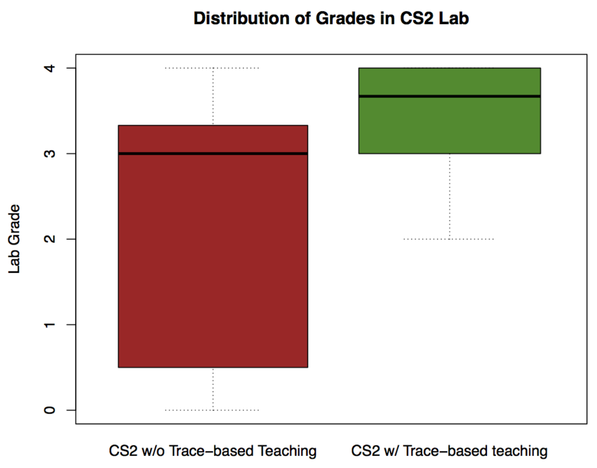
To understand the significance of the grade data, we used an analysis of variance (ANOVA) test. ANOVA examines the variance in the data of two groups to determine the confidence with which one can state that two populations' means are different [12]. We evaluated the student grade data using ANOVA to establish if the grade improvements seen with trace-driven teaching are significant. We found that while the differences in the lecture grades are strong, they are not quite statistically significant ($p = 0.0805$). The same cannot be said for students' lab grades, however. The improvement in these grades is very, very strong and well beyond what is necessary for statistical significance ($p = 0.000131$). Initially, this result may seem odd: the improvements from trace-driven teaching are far stronger for the lab portion of the course rather than the portion that uses traces. We argue this makes sense: the improvement reflects students' improved understanding of the material and holding viable mental models. This improvement has the biggest impact on students' programming skills, the skills most directly assessed, and so most visible, within the laboratory portion of the class.

Finally, since one of our motivations for using trace-driven teaching was the high rates that students had been dropping out or failing our introductory courses, we examine student success rate. Fully one-quarter (25.49%) of students did not successfully complete the CS2 course in the three years prior to this change. In the four years

[3]To validate this assumption, we examined grades in the instructor's other courses. They were unchanged during this time.



(a) While the median lecture grade in CS2 has not changed, the increase in student learning can be seen in the higher skew of the grade distributions. Using ANOVA, the differences are significant with p=0.08.



(b) Even while reducing the time students spend writing code in-class, trace-driven teaching improves students coding ability by increasing their understanding of the material. This change is statistically significant with a p=0.000131.

Figure 3: Graphs showing grade distributions in a CS2 class before and after changing to trace-driven teaching. The boxes show the range of grades in the 1 - 3 quartiles, the dashed lines show the range of grades within 1.5 times of the interquartile range beyond these quartiles, and the solid line is the median grade. As shown by these grades, trace-driven teaching has improved student understanding of the material and their ability to program.

following this change, only 4 out of 57 students (8.51%) have failed this course (no students in this time dropped out). As much as the students' reports of our approach's helpfulness and the increase in student grades, this reduction in the rate in which students did not successfully complete our course shows the value of trace-driven teaching.

5. RELATED WORK

While this is the first paper of which we know that studied the effectiveness of organizing classes around traces, there are other works in related topics such as visualizing a program's state or using traces to assess or improve student learning. We discuss these related works now.

Many papers have been published in which ways of visualizing a program's heap state is discussed. These systems include more general means of visualizing such as Heapviz [1], UUWhistle [15], and visualization tools built into IDEs such as BlueJ [8] and jGRASP [4]. Other systems have been designed to visualize more specific data structures. Examples of these latter systems include iSketchmate [13] and CSTutor [2]. While these papers share our goal of creating means to educate students, they cannot be used with only pencil and paper and do not examine the effectiveness of organizing a class around these topics.

Other research has investigated using traces to assess or improve student learning. An early important work in this area found that introductory students were especially weak at tracing [9]. Unlike our research, this earlier work only looked at students' tracing ability and did not examine the benefits of using traces to educate students. While further research examined the correlation between students ability to trace code with their ability to write code [10], the tracing in that research does not model what occurs in memory and so is an assessment tool and not a way to help students develop viable mental models. Further research by Holliday and Luginbuhl [6] used a trace format created by the authors to document a correlation between students' ability to draw traces and their comprehension of the material. Unlike our research, Holliday and Luginbuhl's trace format did not visualize all of a program's memory and their research did not examine the effectiveness of using traces to teach introductory topics. Thomas, et al., examined the effect of providing students with incomplete traces when asked to answer multiple-choice questions [16]. While some of their findings, that students who draw traces are better able to answer questions, match ours, they did not consider the effect of organizing their course around traces and so could not overcome student resistance. Finally, Ma, et al., used visualizations and cognitive conflict to improve student understanding of value assignments [11]. Like us, they found that visualizations can provide statistically significant improvements in student learning, but only considered the gains from a single exposure to visualizations and did not consider the gains possible when visualization is used throughout a course.

6. CONCLUSION

This paper introduces trace-based teaching which centers teaching introductory programming courses around *program memory traces*. *Program memory traces* present an accurate memory model upon which programming concepts can be explained to students. We show that students feel that tracing code help them learn and hold tracing as more important to their learning and understanding the material than writing code. We also document that trace-based teaching led to improvements in student learning by finding a statistically significant improvement in student programming grades. Thus, this paper shows that program memory traces help students develop viable models of programming concepts and increases the rate students successfully complete the course.

Acknowledgements

The authors wish to thank the anonymous reviewers whose comments helped strengthen this work. They also wish to thank Marge Foster for her help in developing this manuscript.

7. REFERENCES

[1] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th international symposium on Software visualization*, pages 53–62, 2010.

[2] S. Buchanan, B. Ochs, and J. J. LaViola Jr. CSTutor: a pen-based tutor for data structure visualization. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 565–570, 2012.

[3] M. Craig, S. Petersen, and A. Petersen. Following a thread: knitting patterns and program tracing. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 233–238, 2012.

[4] J. H. Cross, II and T. D. Hendrix. jGRASP: an integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond. *Journal of Computing Sciences in Colleges*, 23(1):5–7, Oct. 2007.

[5] T. Götschi, I. Sanders, and V. Galpin. Mental models of recursion. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 346–350, 2003.

[6] M. A. Holliday and D. Luginbuhl. CS1 assessment using memory diagrams. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 200–204, 2004.

[7] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111, 2010.

[8] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), Dec. 2003.

[9] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, 2004.

[10] R. Lister, C. Fidge, and D. Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 161–165, 2009.

[11] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood. Using cognitive conflict and visualisation to improve mental models held by novice programmers. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 342–346, 2008.

[12] 7.4.3. Are the means equal? NIST/SEMATECH e-Handbook of Statistical Methods, Retrieved 16 Aug. 2012. Available at: `http://www.itl.nist.gov/div898/handbook/prc/section4/prc43.htm`.

[13] M. C. Orsega, B. T. Vander Zanden, and C. H. Skinner. Experiments with algorithm visualization tool development. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 559–564, 2012.

[14] J. Sorva. The same but different. In *Proceedings of the 8th International Conference on Computing Education Research*, pages 5–15, 2008.

[15] J. Sorva and T. Sirkiä. UUhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 49–54, 2010.

[16] L. Thomas, M. Ratcliffe, and B. Thomasson. Scaffolding with object diagrams in first year programming classes: some unexpected results. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 250–254, New York, NY, USA, 2004. ACM.