

Using Program Visualization to Illuminate the Notional Machine

by

James A. Juett

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2016

Doctoral Committee:

Assistant Professor Georg Essl, Chair
Professor Mark Ackerman
Lecturer Andrew DeOrio
Associate Professor Chris Quintana

©James A. Juett

2016

TABLE OF CONTENTS

List of Figures	v
Abstract	vii
Chapter	
1 Introduction	1
1.1 Why Do Students Struggle With Programming?	1
1.1.1 Language Constructs	2
1.1.2 Problem Solving	3
1.1.3 Understanding the Notional Machine	4
1.2 Program Visualization	5
1.2.1 The Power of Visualization	5
1.2.2 Program Visualization Tools	6
1.3 Designing Program Visualization Tools	6
1.3.1 Illuminating the Notional Machine	7
1.3.2 Interaction with Dynamic Programs	8
1.3.3 Working with Own Code	8
1.3.4 Effortless Visualization	9
1.4 The Labster System	9
1.5 Dissertation Overview	9
1.6 Summary of Contributions	11
2 Background	12
2.1 Learning Theory	12
2.1.1 Bloom’s Taxonomy of Learning Objectives	12
2.1.2 The Theory of Active Learning	13
2.1.3 Constructivism	14
2.1.4 Constructionism	15
2.1.5 Inductive vs. Deductive Learning	15
2.1.6 Cognitive Load Theory and Learned Schemas	16
2.1.7 Cognitive Theory of Multimedia Learning	17
2.2 Fragile Knowledge in Programming	18
2.2.1 Misconceptions and Mistakes	18
2.2.2 Bugs and Debugging	19
2.2.3 Abductive Learning	21
2.3 Tools For Learning Programming	24

2.3.1	Program Visualization	24
2.3.2	Algorithm Visualization	26
2.3.3	Visual Programming	28
2.3.4	Visual Debuggers	28
2.4	Interaction and Engagement with Visualizations	29
2.4.1	The Engagement Hypothesis and Taxonomies	30
2.4.2	Empirical Results in Learner Engagement	34
2.4.3	Limitations of the Engagement Hypothesis and Taxonomies	36
2.5	Effortless Visualization	39
2.6	Other Trends in Program Visualization	41
3	Contemporary Program Visualization Systems	43
3.1	Systems for Preparing Visualizations	43
3.2	Program Visualization Systems	44
4	Illuminating the Notional Machine	48
4.1	Notional Machines and Black Boxes	48
4.2	Thinking About Dynamic Programs	50
4.3	Mental Models	51
4.3.1	Mental and Conceptual Models of the Notional Machine	52
4.4	Working with Own Code	53
4.5	Techniques for Illuminating the Notional Machine	55
5	The Labster System	57
5.1	Narrative Example of Using Labster and Code-Context Visualization	57
5.2	Situating Labster	60
5.2.1	Ease of Use	60
5.3	Editor and Compilation	60
5.4	Visualization	61
5.4.1	Interaction Modalities	62
5.4.2	Navigation and Controls	63
5.4.3	Navigation Labster Functions	64
5.5	Runtime Feedback	64
5.6	Novelty	65
6	Experiments and Methodology	67
6.1	Participants and Data Collection	67
6.1.1	Grouping by Pre-Lab Confidence Levels	67
6.1.2	Grouping by Time Spent	68
6.1.3	EECS 280 Terms	68
6.2	Methodology	69
6.2.1	Lab Exercises	70
6.2.2	Pre and Post Lab Surveys	70
6.2.3	Student Confidence	70
6.2.4	Questions about Lab Efficacy	72
6.2.5	Questions about Using Labster	72

6.2.6	Conceptual Questions and Coding Problems	73
6.3	Measurement, Statistical Procedures, and Significance Testing	73
7	Experiment: Expression Evaluation	74
7.1	Background	74
7.1.1	Visualization Systems for Functional Programming	75
7.2	<i>in situ</i> Expression Evaluation in Labster	76
7.3	Experiment	78
7.4	Results	80
8	Experiment: Recursion	83
8.1	Background	83
8.1.1	Mental Models of Recursion	83
8.2	Control Flow and Subcall Execution in the Notional Machine	84
8.3	The <i>Static Source</i> Model in Contemporary Systems	85
8.4	The <i>Code Stack</i> Model in Labster	87
8.4.1	Contrasting the <i>Code Stack</i> and <i>Static Source</i> Models	89
8.5	Experiment	90
8.5.1	Labster Blue and Labster Maize	93
8.6	Results	94
9	Experiment: Navigation Controls	99
9.1	The Role of Navigation in Program Visualization	99
9.2	Navigation Features in Existing Systems	100
9.3	Varied Navigation Controls in Labster	101
9.4	Experiment	102
9.5	Results	103
9.5.1	Post-Lab Confidence	103
9.5.2	Post-Test Performance	104
9.5.3	Labster Evaluation and Time Spent	105
9.6	Summary	105
10	Discussion	108
10.1	What Kinds of Learning Outcomes are Affected?	108
10.1.1	Conceptual Recall	108
10.1.2	Code Reading	109
10.1.3	Code Writing	109
10.1.4	Student Confidence and Subjective Evaluation	111
10.2	The Impact of Student Confidence	112
10.2.1	Adding Training Wheels to Labster	113
10.3	Code-Context Visualization in Labster	114
11	Conclusion	116
	Bibliography	118

LIST OF FIGURES

1.1	Example of Buggy Code	2
1.2	General Principles of Program Visualization	7
2.1	Brief Example of the & Operator	22
2.2	Comprehensive Example of the * and & Operators	23
2.3	General Categories of Software Visualization Systems	25
2.4	The Original Engagement Taxonomy	31
2.5	Direct Engagement and Content Ownership in the 2DET	32
2.6	Trends of Engagement in the 2DET	33
2.7	Studies of <i>Epistemic Fidelity</i>	37
4.1	Illuminating the Notinal Machine - The Big Picture	49
4.2	Aspects of Educational Program Visualization	56
5.1	A Recursive Implementation of the Factorial Function	58
5.2	Visualization of the Factorial Function	59
5.3	Compiler Error Messages in Labster	61
5.4	Additional Compiler Error Messages in Labster	62
5.5	Runtime Error Messages in Labster	65
6.1	Pre-lab Confidence Distribution	68
7.1	Example of Expression Evaluation Visualized in Labster	78
7.2	Example of Expression Evaluation Visualized in Jeliot 3	79
7.3	The code-tracing task from the Arrays/Pointers lab surveys.	79
7.4	Code-Tracing Performance	80
7.5	Students' Self-Assessment	81
7.6	Students' Perceived Efficacy of Lab Exercises	81
7.7	Students' Evaluation of Labster	82
8.1	Example of Subcall Execution in the Online Python Tutor	85
8.2	Example of Subcall Execution in the Jeliot 3	86
8.3	Example of Subcall Execution in the Teaching Machine	87
8.4	Example of Subcall Execution in Labster	88
8.5	Parameter Passing and Returns in Labster	89
8.6	Code-Writing Task 1	91
8.7	Code-Writing Task 2	92

8.8	Labster Blue and Maize	93
8.9	Students' Performance on Code-Writing Task 1	94
8.10	Students' Performance on Code-Writing Task 2	95
8.11	Students' Confidence Levels	95
8.12	Students' Perceived Efficacy of Lab Exercises	96
8.13	Students' Evaluation of Labster	97
8.14	Students' Performance on a Recursion Midterm Question	98
9.1	Navigation Controls in Labster	102
9.2	Distribution of Confidence Levels	103
9.3	Time Spent Working with Labster	104
9.4	Students' Performance on Code-Writing Task 1	104
9.5	Students' Performance on Code-Writing Task 2, Confidence Group 3	105
9.6	Students' Performance on Code-Writing Task 1, Confidence Group 4	105
9.7	Students' Perceived Efficacy of Lab Exercises	106
9.8	Students' Evaluation of Labster	107

ABSTRACT

Using Program Visualization to Illuminate the Notional Machine

by

James A. Juett

Chair: Assistant Professor Georg Essl

Programming is an essential skill in computing and engineering fields, but many students are not competent programmers even after completing initial programming courses. They have difficulty writing correct code, and the skills they develop often turn out to be fragile and do not generalize well to dealing with new problems. A key contributor to this issue is the lack of “hands on” experience with the *notional machine* - the conceptual computer on which their programs run. For many students, the computer is essentially a black-box, and at best they receive very indirect feedback about what each part of their code actually does at runtime.

We attack this problem using interactive program visualization, which illuminates the notional machine and empowers students to have meaningful, hands-on experiences with what their own code actually does at runtime. In this dissertation, we investigate design choices that go into creating a program visualization tool to best support this approach. In particular, we explore which conceptual models should be presented for expression evaluation and subcall execution, what navigation controls should be offered, and how to provide feedback when problems occur at runtime.

The contributions of this work are novel solutions to these questions and a large-scale empirical study in an authentic educational setting that gives evidence they are effective. We also present Labster, a web-based C++ program visualization system, which brings together our innovations as well as successful techniques from the previous literature into a working ensemble. Labster has proven effective as both an educational tool and a platform for program visualization research.

CHAPTER 1

Introduction

Many students are not competent programmers even after completing initial programming courses. On both language-independent and language-specific tests [1] designed to measure proficiency with specifically those topics commonly taught in CS1 courses, students performed poorly. In addition, failure rates in these courses are concerning. [2]

Students have difficulty reading and correctly understanding what code does (i.e. *program comprehension*). In a large-scale, multi-national study [3], students who had completed an introductory course were tested on their ability to predict the behavior of code or to select the correct completion for a partial code snippet. Average performance was disappointing and indicates many students either do not possess the ability to mentally trace code or cannot do so with sufficient accuracy/precision.

Several studies have also established that students perform poorly in code writing or design tasks (i.e. *program composition*) [4]. Even when the task is relatively simple, such as averaging a sequence of numbers (often known as the "Rainfall Problem" following Soloway's original formulation [5]), many students are unable to compose a correct program to solve the problem on the first try [6]. Knowledge and skills required for program comprehension and composition are linked [7], and students' ability to write programs relies upon their ability to trace and explain what code does [6].

1.1 Why Do Students Struggle With Programming?

We will discuss some of the contributing factors to students' difficulties with programming beginning with two common themes in the literature: students' understanding of programming language constructs and students' problem solving skills. To focus our argument, we consider the specific implications of these themes for a characteristic example of students' buggy code originally presented in [8] (see figure 1.1). In this example, a student has incorrectly implement a guard for invalid input and their code ends up trying to print a result even though none could be computed.

```

void func(int income, char maritalStatus){
    int tax;
    if (maritalStatus == 'm'){
        tax = married(income, tax);
    }
    else if (maritalStatus == 's'){
        tax = single(income, tax);
    }
    else{
        cout << "Invalid marital status." << endl;
    }

    // Oops! Prints even when invalid!
    cout << "Total tax: " << tax << endl;
}

```

Figure 1.1: Adapted from Spohrer and Soloway [8]. A buggy implementation of a function that should compute and print income tax differently based on marital status. If the marital status is ‘m’ or ‘s’, an appropriate function should be called to compute the tax and then it should be printed. However, if the marital status has any other value, an error message should be printed instead of the total tax. This program behaves incorrectly because it attempts to print the total tax even if the input was erroneous.

1.1.1 Language Constructs

Many introductory programming textbooks focus primarily on the details of a particular programming language, often structured and organized around particular language constructs [9]. It seems clear at least that the outcome of understanding language constructs is desirable, because they are the basic building blocks available to students for writing programs.

However, the real question at hand is whether knowledge of language constructs (or lack thereof) is a major factor in students’ difficulty with programming. The evidence suggests this is not the case. Soloway and Spohrer [8] analyzed a collection of students’ buggy solutions to simple problems and found no empirical support that most bugs come from misunderstanding language constructs. An international, multi-institution study by Lister et al. is in line with this finding [3].

Neither study investigated particularly complex language constructs, but the findings do establish novice programmers’ problems are not simply due to construct misconceptions. For example, in the marital status program, students may completely understand the semantics of an if statement, but still not catch that the output statements were outside the guard. This is not to say learning about language constructs is not important or should not play a major part in introductory

courses, but rather that the evidence points to other problems causing difficulty for programming students.

1.1.2 Problem Solving

A potential explanation for students' inability to compose correct programs even if they understand individual language constructs is that they may lack the problem solving skills prerequisite for formulating a correct solution. Essentially, in order to write a program to solve a problem, students must first use generic problem solving skills to formulate the problem as an algorithm. This is not necessarily a skill that is innately possessed and that may need greater attention in introductory courses [10, 11]. Soloway [5] suggests teachers should explicitly teach students to correctly identify and isolate individual *goals* their code must achieve and should help them develop a toolkit of effective *plans* ("stereotypical, canned solutions") for realizing those goals.

However, considering that students have trouble programming solutions to even very simple problems like the marital status example or averaging a sequence of numbers, the difficulties they face cannot be purely a problem solving issue. The student certainly understands a correct solution should not perform the same actions on invalid input. Most would likely realize what the problem was if verbally walked through the execution of their algorithm on problematic input, but for some reason were unable to detect the fault when writing the code.

Neither issues with language constructs or problem solving alone prove a satisfactory explanation for programming students' difficulties. To solve a problem with a program they must not only formulate their solution strategy, but also translate it to an algorithm embodied as sequence of programming constructs used together and in precisely the proper sequence. Even while it may be the case that students appear to understand individual language constructs in isolation, more subtle misconceptions or a lack of practice with putting them together become problematic. It seems that even when beginning programmers can conceive workable solutions to a problem, the real difficulty they have is mapping that solution to a working program or detecting flaws as they are writing the code.

This observation brings us to the idea that students may not have a viable mental model of the notional machine – a conceptual understanding of how programs actually work. Without this key ingredient, students' problem solving skills and knowledge of language constructs are irrelevant - they may be able to work out a solution in the problem space, but they can't conceptualize or translate their solution to a program that implements the solution because they don't have an effective understanding of the artifact (i.e. the notional machine) they are programming.

1.1.3 Understanding the Notional Machine

In programming, a “notional machine” is an idealized, hypothetical machine on which programs run [12]. A notional machine serves as an abstraction to explain the way programs execute, and a viable mental model of the notional machine is a prerequisite for both program comprehension and composition tasks. However, students do not innately possess an effective model of a computer [13]. In programming, there is often no clear “real-world” parallel to a particular concept, and the analogies students tend to apply by intuition often lead to non-viable mental models. Even as they learn, novice programmers’ mental models of a notional machine are likely to be “incomplete, unscientific, deficient, lacking in firm boundaries, and liable to change at any time.” [14].

As such, developing a viable model of the notional machine must constantly be kept in sight as one of the objectives for students in programming courses. Unfortunately, it seems that present teaching strategies are not leading students to develop adequate mental models of the notional machine, as they perform poorly in code comprehension tasks like tracing program execution or checking correctness [3]. These skills are prerequisite to effectively solving problems by writing programs, and in particular are essential for translating an algorithm into a working program.

The core problem is that many students only experience the notional machine as a “black-box”. If they only learn individual language constructs in isolation, students fail to gain an appreciation for the notional machine that defines the way constructs work together as components in a program as a whole. Furthermore, a student in a traditional setting writes a program, attempts to compile it, and runs it - the only pieces of feedback they receive are compiler errors/warnings and program output. Neither of these provide much, if any, insight into what a program actually does when it runs. At best, students are left trying to infer dynamic properties of their programs from static, textual code and verbal descriptions of what it should do. Without a window into what is happening at runtime, students’ construction of mental models of the notional machine is hindered.

Ultimately, problem solving should not be done solely in order to produce program code that generates the correct output for particular input cases. Beginning programmers may find a certain degree of “success” with this approach, but without an appreciation for the dynamic program defined by their source code, students will develop fragile schemas for program construction. That is, they may find a workable solution after rearranging constructs a few times, but this almost certainly will not generalize to an approach that works well for novel programs. The end goal of programming education, at least, has never been working program code - it has been the development of mental models and skills to comprehend and write working programs in the future.

Our goal is thus to find the most effective ways for students to “get to know” the notional machine intimately, in the spirit that du Boulay [15] expressed:

“A running program is a kind of mechanism and it takes quite a long time to learn the

relation between a program on the page and the mechanism it describes. It's just as hard as trying to understand how a car engine works from a diagram in a text book. Only some familiarity with wheels, cranks, gears, bearings, etc. and "getting one's hands grubby," gives the power to imagine the working mechanism described by the diagram and crucially, to relate a broken engine's failure to work to malfunctions in its unseen innards."

1.2 Program Visualization

In what follows, we discuss the approach of using program visualization to give students "hands on", interactive experiences with the notional machine that aid them in knowledge construction and becoming more proficient programmers.

1.2.1 The Power of Visualization

Most computer science educators regularly use visualizations in their teaching and believe in the power of those visualizations to augment traditional learning [16]. A visualization can provide richer resources for learning than text alone, and can be more intuitive to students if it leverages metaphors students are familiar with (e.g. using an arrow to represent a pointer, representing an array as spatially consecutive elements). Students are also encouraged to draw out diagrams on their own when tasked with understanding a complex piece of code. For example, memory diagrams are a standard tool for illustrating complex relationships between different parts of a program's state.

Visual representations can also help to manage complexity for the novice programmer. If a programmer is attempting to mentally trace through code execution, they must keep track of many different aspects of a dynamic program including the function call stack, which statement is currently executing, and the values of variables. Many of these may not be immediately relevant, but still must somehow be tracked to keep a mental simulation coherent. Keeping track of all this information puts a serious strain on the capacity of human working memory, especially for beginning students. Both text-based and visual representations can be used to help students track information while tracing through code, but visual approaches can additionally leverage the strength of human visual processing. The old adage, "A picture is worth a thousand words" comes to mind.

When instructors use diagrams to illustrate what is going "inside the computer" when a program runs, their essential goal is to go beyond the source code and give students a peek into the notional machine. This lifts the space for discussion, learning, and problem solving away from the source code. Schemas that students develop for understanding or writing programs at this level

will almost certainly exhibit fragility in the general case, since patterns in source code do not line up with patterns in behavior. Instead, students should be thinking about their code as instructions for what the notional machine should do at runtime. An understanding of programs in this context will be much more robust.

1.2.2 Program Visualization Tools

Program visualization tools aid in the production of visualizations relevant to programming concepts. For example, a system may automatically generate a visualization of a running program that a teacher would regularly have to laboriously draw out by hand. Many different varieties of visualization systems exist (see e.g. [17],), but our work is focused on generic program visualization systems as described in Sorva’s review [18]. This kind of system can visualize most any code written in its target language and is able to give a holistic view of the notional machine as it dynamically executes program code. We also focus on systems that are highly interactive and support visualization of students own code. These systems fundamentally transform the experience students have when learning to write programs by providing an environment where running their code is a truly “hands on” experience with the notional machine.

The benefit of “hands on” experience is recognized in many fields. A surgeon cannot effectively learn to operate on real patients just by reading books or listening to an expert’s description of the process. Likewise, it is difficult for students to master programming simply from a written description of what each programming language construct does. Instead, students should learn from direct experiences with the notional machine. We can crystallize this idea through a constructivist lens - students are not passive receivers of knowledge, but rather must be active participants and construct knowledge as a reaction to the experiences they have. A written description of how a piece of code is executed is little more than a second hand account of someone else’s understanding of the notional machine. Experiences with words describing the notional machine is much less rich a material for knowledge construction than access to the notional machine itself.

1.3 Designing Program Visualization Tools

A well-designed program visualization tool must provide students with interactive, “hands-on” experiences from which they can construct an understanding of programming. They must be allowed to approach problem solving through programming as an activity focused on the behavior of the notional machine and not just as an exercise in writing source code. To support this, the tool must illuminate the notional machine by making clear precisely how the dynamic program works at runtime - turning the black-box into a glass-box. It is essential for students to actively engage with the

visualization and to be able to work with code they themselves write. Finally, the process must be “effortless” and not burden students with any unnecessary overhead that might hinder the learning process. These principles are summarized in Figure 1.2, discussed briefly below, and explored at length throughout this dissertation.

Principle	Description
Illuminate the Notional Machine	A visualization must make clear precisely how each part of a dynamic program executes and how they work together at runtime - turning the black-box into a glass-box by illuminating the notional machine.
Interaction with Dynamic Programs	The visualization must provide an interactive, “hands on” experience with dynamic programs and should be used in a way that encourages students to actively engage in learning.
Work with Own Code	Students write and work with their own code to support an individualized learning experience that is intimately tied to their current understanding, and they are able to refine that understanding based on what they see and do in the visualization.
Effortless Visualization	No extra overhead required to run, visualize, or interact with programs, which might otherwise hinder students’ learning.

Figure 1.2: General principles of a successful approach to program visualization pedagogical approach for learning about the notional machine via interactive program simulation. Each is necessary for a high-quality learning experience and should be addressed by individual aspects of a program visualization.

1.3.1 Illuminating the Notional Machine

In a traditional setting, the notional machine is essentially a black box to students. The limited feedback they get from program output (or error messages if something goes wrong) does little to help them learn about what their program is actually doing at runtime. Students essentially face a prohibitive attribution problem - they don’t know why their program does or doesn’t work because they can’t see what the code is actually doing. We seek to eliminate this problem by illuminating the notional machine, so that the black-box becomes a glass box. Specifically, program visualization provides a way to give students interactive, “hands on” experiences with conceptual models of the notional machine, which they use in turn to construct their own understanding.

The notional machine is a complex artifact, and the design of a program visualization system involves several choices about how to best illuminate the way it works. The primary goal of this dissertation is to investigate the design choices that go into creating an interactive program visualization to illuminate the notional machine. In particular, we explore which conceptual models should be presented for expression evaluation and subcall execution, what navigation controls

should be offered, and how to provide feedback when problems occur at runtime. We discuss these issues at length throughout this work. In particular, section 4 further explores the goal and general strategies for illuminating the notional machine and sections 7-9 describe our experimental investigations into specific techniques.

1.3.2 Interaction with Dynamic Programs

In addition to opening up the black-box so that students have a clearer picture of the notional machine, we also strive to give them the opportunity to interact with their programs as they run on the notional machine. The importance of this is grounded in the theories of active learning and constructivism – students must actively engage with material and construct an understanding based on their experiences. Learner engagement and interaction is also one of the most discussed topics in the algorithm and program visualization literature. As expressed in the influential report of a working group at ITiCSE 2002 [16], "...visualization technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity."

Much research work concerning engagement with educational visualizations has been structured around proposed taxonomies of engagement [16, 19, 20, 18] that frame certain kinds of engagement into categories. Experiments (see e.g. [21, 18]) have provided support for the core idea of the engagement hypothesis - namely that viewing a visualization is better than not viewing one at all, and that it is important for students to actively control the visualization. Higher levels of engagement (e.g. responding, applying, constructing) have also been found more effective in certain contexts, but there is not enough evidence to conclude they are categorically better.

We discuss interaction with program visualization and learner engagement further in section 2.4.

1.3.3 Working with Own Code

A major component of our approach is that students are able to perform any visualization and interaction activities with code that they themselves have written. Students will be fundamentally more invested in content they have had a hand in generating, and the experience they have with the visualization is responsive to their current mental model which was used to write the code. Misconceptions that students hold will be exposed when their code is run and are able to refine their understanding in response. Allowing students to freely work with their own content also enables them to explore what-if questions as they come up and removes the sole reliance on an instructor preparing a small set of examples ahead of time that must somehow be relevant to all students.

We discuss the implications of students working with their own code further in section 4.4.

1.3.4 Effortless Visualization

One of the main obstacles to the adoption of visualization tools in mainstream education is the large amount of time that may be required to produce quality visualizations, as well as the amount of overhead work required for students to use them [16]. As such, a design goal of any program visualization system should be to minimize the amount of up front time and work that must be invested for the visualization can be used. A system is called “effortless” if it achieves this goal. A thorough discussion of effortlessness in existing systems and throughout the literature is given by Ithantola et al. [22] who also describe a taxonomy of effortless creation of algorithm visualizations.

It is important to stress that an effortless approach to program visualization does not and should not attempt to remove the need for “effort” on the part of the learner. Active engagement, which is essential for high quality learning, involves a great deal of effort! Rather, the effortless approach is to eliminate unnecessary effort required to use a system, but not related to the learning process. Once this is accomplished, students can focus their effort on the most important aspects of the learning process. In particular, students must be able to work with and visualize their own code without having to do any extra work (e.g. adding annotations to code, setting up the visualization, etc.).

We discuss the literature on effortless visualization further in section 2.5.

1.4 The Labster System

In order to evaluate our approach, we needed a concrete realization each of the techniques mentioned above. As such, we designed the “Labster” program visualization system. In short, Labster is a web-based program visualization system that supports students visualizing and interacting with their own code in the c++ language. No installation is required, and using it is as simple as visiting a web-page. Labster serves both directly as an educational tool, but also as a research platform to evaluate both our overall approach and individual techniques for illuminating the notional machine.

We have integrated interactive exercises using Labster into the coursework for the EECS 280 - “Programming and Introductory Data Structures” course at the University of Michigan. We report results from an ongoing evaluation in this context through a large-scale between subjects quasi-experiment. Students’ response to the system has been positive, and results show using the tool has a significant impact on students’ learning outcomes.

1.5 Dissertation Overview

The unifying and guiding research question for this work is:

“How do design choices for illuminating the notional machine in interactive program visualization systems affect students learning outcomes in introductory programming courses?”

The primary goal of this dissertation is to investigate the design choices that go into creating an interactive program visualization with the primary goal of illuminating the notional machine. In particular, we explore which conceptual models should be presented for expression evaluation and subcall execution, what navigation controls should be offered, and how to provide feedback when problems occur at runtime.

In chapter 2, we describe background in learning theory, programming education, and program visualization that is relevant to our work. In chapter 3, we give a brief overview of several contemporary program visualization systems.

In chapter 4, we further motivate the goal of learning about the notional machine and its implications for program visualization, as well as discuss our broad approach to investigating design choices for program visualization systems in order to better illuminate the notional machine. In chapter 5, we provide a brief overview of the Labster program visualization system, which was created as part of this work and in order to support our experimental investigations. In chapter 6, we describe our experimental methodology.

In chapter 7, we present a novel approach to visualizing expression evaluation “in situ”. Since a major function of the notional machine is to perform computations by evaluating expressions, it is important for a visualization system to present a conceptual model of expression evaluation that shows the fine details of each computation in an intuitive fashion. Our approach is based on a term-rewriting model, but also brings the source code for the expression itself to life as a dynamic visualization in the context where students will actually encounter expressions. We also describe an evaluation the effectiveness of this approach for improving students code-tracing skills.

In chapter 8, we investigate the decision of which conceptual model to present for function calls and the call stack, and how to represent them visually. To ground our discussion, we review the literature on students’ understanding of recursion and active/passive control flow. We then present the novel *codestack* model used to represent the call stack in Labster and an experimental comparison with the *static source* model used in most contemporary program visualizations systems.

In chapter 9, we explore the the impact of the kinds of navigation controls provided by a visualization system and how they affect student learning. Considerable evidence supports the general idea of allowing students to control a visualization, and many contemporary systems support this to some extent, but little work has been done to experimentally investigate the learning impact of specific mechanisms for navigation. We describe initial results from an empirical evaluation of

two different versions of the Labster program visualization system, one with very basic controls and another with full featured navigation.

In chapter 10, we summarize and review the findings of this dissertation and discuss directions for future work.

1.6 Summary of Contributions

A brief summary of the contributions of this dissertation.

- The Labster system, a web-based, interactive program visualization system for the C++ language that supports students working with their own code. Labster has proven effective as both an educational tool and a platform for program visualization research.
- Several novel approaches to program visualization, including “in-situ” expression evaluation, the *code stack* model of subcall execution, “point+click” navigation, and static/dynamic analysis for improved feedback. These, as well as other successful approaches from the previous literature are realized as an ensemble in the Labster system. (See figure 4.2.)
- An approach that emphasizes illuminating the notional machine as one of the major principles of effective program visualization alongside interaction, working with own content, and effortlessness. (See figure 1.2.)
- A large-scale empirical study in an authentic educational setting that supports our approach. Our key results include:
 - Students working with Labster in lab activities saw improvements in confidence and code-tracing skills over students who performed the same tasks without visualization, and also felt that the work they did was more effective.
 - The *code stack* model used in Labster improves learning outcomes compared to the *static source* model used in many contemporary program visualization systems.
 - A system with full-featured navigation controls drastically improves students subjective experience with program visualization and leads to improved learning outcomes as compared to a system with very basic navigation controls.

CHAPTER 2

Background

2.1 Learning Theory

Throughout this work, we refer to several theories of learning. This section provides a brief overview and introduction to these theories, in particular as they are relevant to programming education and pertinent for the design of program visualizations.

2.1.1 Bloom's Taxonomy of Learning Objectives

One of the most foundational works in the educational literature is Bloom's Taxonomy of Educational Objectives [23], originally published in 1956. The taxonomy describes six different objectives within the cognitive domain, in increasing order of complexity and abstractness: *knowledge*, *comprehension*, *application*, *analysis*, *synthesis*, and *evaluation*. It is assumed that students will generally approach and achieve the objectives in an ordered fashion, with the higher-level objectives required for achieving the fullest mastery of a topic.

In 2001, a substantial revision to the taxonomy was published [24]. The revised taxonomy is two-dimensional and separates out the kind of knowledge under consideration (*factual*, *conceptual*, *procedural*, and *metacognitive*) from the original cognitive process dimension. The categories in the cognitive process dimension were renamed using a verb form to more precisely express their relation to what kind of learning activity the learner is participating in. The new categories of this dimension are *remember*, *understand*, *apply*, *analyze*, *evaluate*, *create*. The create category (previously synthesis) was moved to the highest-level objective, replacing evaluation. In both the original and revised taxonomies, the six levels of the cognitive process dimension were further subdivided into 19 subcategories, each of which represents a more specific kind of learning objective. For example, evaluation is divided into *checking* and *critiquing*.

The taxonomy can be used as a lens to understand the learning objectives as they appear specifically in programming education. In the knowledge dimension, factual knowledge may concern

individual aspects or building blocks of a program (e.g. language constructs, whether a language is compiled or interpreted, properties of datatypes), whereas conceptual knowledge involves the relationships between them and the way they work together in a program as a whole. The latter also involves an understanding of a program as a dynamic entity at runtime. Procedural knowledge involves the processes and skills students will develop as they become more practiced programmers and internalize strategies for solving common types of problems. Finally, students may approach metacognitive objectives by reflecting on the way they think about programming, for example by explicitly identifying and codifying programming strategies they use.

We will not discuss each of the 19 specific categories along the cognitive process dimension here, but we provide a few examples to illustrate the kinds of learning objectives that might appear in programming education. Lower-level objectives at the remembering level include the semantics of individual constructs in isolation (e.g. which operator performs what operation), or at the understanding level, classifying constructs as either expressions, statements, or declarations. Carrying out mental traces of program code exemplifies the application level. Higher-level objectives tend to match the general activities one expects an expert programmer to conduct in practice. Debugging essentially amounts to analyzing code piece by piece, while evaluating might entail judging code on style or computational efficiency. Finally, writing programs is of course an exercise in creating. As with the taxonomy in general, low-level objectives in programming education must often be mastered before higher-level objectives are approachable. For example, experienced programmers often rely on mental traces of code when trying to understand and write programs.

Much has been said about Bloom's Taxonomy and learning objectives in computer science courses and their design (see e.g. [25, 26, 27]). Bloom's taxonomy also informs the design of the engagement taxonomies, [16] which we discuss further below. Other work attempts to assess whether Bloom's Taxonomy can be applied to computer science [28] and a taxonomy for learning objectives specifically tailored to computer science education has also been proposed [29] that takes into account particular emphases on things like the creation of tangible artifacts (i.e. programs).

2.1.2 The Theory of Active Learning

The theory of active learning frames the learning process as one in which the student is an active participant rather than just a passive receiver of knowledge. In particular, higher order learning objectives like analyzing, evaluating, and creating require the learner to take on an active role. The benefits of active learning have been studied extensively in classroom settings and are well supported by empirical evidence [30]. Furthermore, increases in the number of students who understand a particular topic after incorporating active learning can be quite pronounced (e.g. [31]).

While much of the research on active learning has focused on classroom settings, it is reasonable to think the underlying principles are applicable to electronic resources such as program visualizations in whatever setting they are used. In particular, much of the literature on engagement in algorithm and program visualization (see e.g. [16]) is rooted in the theory of active learning.

2.1.3 Constructivism

The fundamental assumption of constructivism is that the learning process is one in which knowledge and understanding are actively constructed by a learner – it is not enough to receive and store information. Learning experiences and the way they are perceived by a student are essentially the raw materials out of which knowledge is constructed. The goal of constructivist learning is to allow students to incrementally refine their understanding toward viability. The practical application of constructivism to education is to design educational and instructional materials that facilitate a learner’s construction of their own viable mental model.

The knowledge construction process works incrementally as a student combines new information and experiences with their existing cognitive framework. As each student brings a different background, their construction process is necessarily unique. It is essential to provide individualized learning experiences that are responsive to where each student is coming from and which allow their understanding to grow and reshape to accommodate new information and insights. The use of a program visualization tool has the potential to affect this process, including in particular the representations they use to understand how program execution works [32].

An instructor’s role in constructivist education not simply providing students with raw materials (i.e. information), but guiding their construction process. This involves the presentation of particular material, examples, practice problems, etc. in order to best lead students to tease out the misconceptions in their own mental models. This requires attention to each student as an individual because each student has different background knowledge and personal learning style. Every student will struggle with a unique set of misconceptions, and some students will take longer to arrive at a viable model or require more guidance to do so than others. A major challenge in modern education is providing this sort of guidance in a way that scales. Program visualization provides a step in the right direction because it allows students to receive much richer feedback about what code does without direct intervention by an instructor. If students are able to have interactive, “hands on” experiences with the artifact they are leaning about (i.e. the notional machine), they can often discover much on their own that a teacher would otherwise have to explain.

2.1.4 Constructionism

Papert's theory of constructionism [33] suggests that students learn as builders of artifacts. Hands-on experience with the construction of an external entity naturally supports a student's internal construction of knowledge. Program visualization tools provide a natural context for this sort of work to occur as students can iteratively write code and test it out in a live environment. On the other hand, students completing traditional programming exercises on a black-box computer are only able to look at the entity (i.e. the program) they are constructing from a very limited perspective. They are only allowed to examine their creation as static code, but not in its true form as a dynamic program executing at runtime.

2.1.5 Inductive vs. Deductive Learning

Educational activities or instructional techniques can often be classified as either deductive or inductive. In a deductive approach, students are first presented with general principle (e.g. a theorem) and then given examples to illustrate the application of that principle. In an inductive approach, on the other hand, students first observe specific phenomena or attack concrete problems which motivate and eventually lead into the discussion of general principles. While tradition and current practice both seem to favor the deductive approach, research has shown inductive methods are often more effective [34].

Active learning is often a necessary component in an inductive approach - the investigative process involved is inherently active. Inductive methods are also in line with constructivist thinking as they present the student with particular experiences that allow them to build up and refine their understanding (rather than a deductive prescription of a "correct" way to think about something). Inductive learning can also alleviate the tendency of students to question why a particular topic is important because the progression beginning with concrete problems or examples illustrates the need for more abstract conceptual principles to deal. Students are more motivated to learn a concept when they clearly understand why it is important (see e.g. [35, 36]).

An approach in which students interact with a visualization of programs they write involves inductive learning. Consider a student whose end goal may be to develop the skills needed to solve problems that require array traversal and manipulation. First, the student may be tasked with writing a simple loop that prints out each element in the array. Then, they may move onto more complex problems like adding up or averaging all the elements in the array. From these exercises, students will begin to develop an intuition (i.e. schemas) for understanding and writing code to traverse loops. They may induce patterns from their experience, such as the accumulator pattern for combining elements. These patterns will be more deeply meaningful since the student discovered it on their own to solve a problem and inherently understands why it was needed.

2.1.6 Cognitive Load Theory and Learned Schemas

Cognitive Load Theory [37, 38] provides insight into the role human information processing and working memory play in the learning process, in particular as limited mental resources that must be used wisely to maximize the effectiveness of learning and teaching. Limits on working memory are well known - the most prominent result being the “magical number seven, plus or minus two” [39] which denotes the number of elements that can be held in working memory at a given time. Cognitive load has also been studied in programming contexts. For example, in one study [40] 30% of students’ debugging breakdowns were tied to attentional problems such as loss of situational awareness or working memory strain.

Strategies exist for coping with these cognitive limitations. Learned schemas allow thinkers to group together many individual pieces of information into “chunks” (i.e. meaningful aggregate elements), because the essential limit is on the number of elements that can be held in working memory and not necessarily on their complexity. Abstraction plays an important role in that only certain high-level aspects of complex elements need to be actively considered for solving a particular problem. In order for an abstraction to be successful, a chunk must consist of information that cohesively forms together as a meaningful whole.

It is fitting to note the unique role abstraction plays in the field of computer science and programming in particular. It appears both as lens for viewing chunked concepts and also directly as a programming technique [41]. Layers upon layers of abstraction are required to approach many problems in the field. For example, one may write code to implement a pathfinding algorithm as a combination of simpler algorithms and operations on abstract data types built up as meaningful groupings of data/operations, which are in turn implemented using a high-level programming language that abstracts the details of machine code, which in turn is another level of abstraction on top of several more dealing with computer architecture, digital circuitry, etc. Learned schemas also inform the way an experienced programmer would approach solving this problem, in particular the abstractions selected for both data and procedures.

Expert programmers appear to be more well equipped to use these kinds of tools to manage cognitive load when programming. An experiment by McKeithen et al. [42] found that expert programmers were able to recall far more information than novices after briefly examining meaningful program code. This suggests experts may have better developed schemas that allow them to organize code into meaningful chunks as they read it. Other works provides evidence that the organization of programming concepts differs between novice and expert programmers [43], and that expert programmings tend to organize concepts in an abstract semantic way while novices rely on syntax [44]. Expert programmers also make use of *beacons* [45], which are recognizable features in program code that can be used to identify a particular technique or algorithm (e.g. a two way swap, the accumulator pattern). An essential consideration in programming education

should be how to best support learners in their creation of these kinds of meaningful and effective schemas, abstractions, and procedural skills that experts use to aid in program comprehension and composition.

Both inductive and deductive approaches can be taken to help students develop learned schemas. Soloway [5] recommended the explicit teaching of *goals* and *plans*, common problems code must solve and "stereotypical, canned solutions" for solving them. However, this deductive approach can be problematic. An instructor can try to explain to students the identifying characteristics of each goal and the rules for applying a corresponding plan, but the understanding will not be as intuitive or as deeply held as if students had inductively "discovered" the goal/plan pair for themselves. Essentially, schemas cannot be transferred from a teacher to learners - they must be constructed by the learner themselves. Playing along with the metaphor, each student must do the "canning" process on their own rather than buying a canned plan off the shelf. Of course, students should still be encouraged to think metacognitively about the libraries of goals and plans they are developing as they learn.

2.1.7 Cognitive Theory of Multimedia Learning

The cognitive theory of multimedia learning (see [46]) informs the design of electronic educational resources to best accommodate the underlying cognitive processes by which people learn. The three major assumptions of the theory are that people possess dual channels for processing textual/verbal information, that the channels have limited capacity, and that the learning process is one in which people actively engage in cognitive processing of received information.

According to the limited capacity assumption, while multimedia can be more effective than text alone, the inclusion of extra media can potentially contribute to cognitive overload of the learner's limited processing capacity. Several types of cognitive overload and suggestions for alleviating each are discussed in [47].

Following the active engagement assumption, the design goal of electronic resources should first and foremost to give learners the tools to actively construct their own mental model of the concepts they are learning. For example, evidence shows that an influential factor in the success of algorithm visualization software is whether the visualization engages students in an active learning activity [48, 16]. However, multimedia do not always encourage active learning and can even be detrimental in some cases [49]. Embedded audio/video are susceptible to this problem if they allow a student to zone-out or are presented in ways that strain a student's attention span.

It is best to approach the incorporation of multimedia from a learner-centered, rather than technology-centered, perspective [50]. From a technology-centered perspective, the focus is on finding ways to use technologies to present educational material. While this may often seem to

be a positive contribution on the surface, it does not always translate to better learning, especially if the technology does not promote active learning. In learner-centered design, the design starts with an understanding of the cognitive processes by which people learn and only incorporates technologies as it aids in those processes. As such, the question to ask when including multimedia in educational resources is not “How can we best represent this concept?” but rather “How can we best help the learner to construct their own coherent understanding of the concept?” The implication for program visualization is that we should not visualize something just because we can, but should instead consider which conceptual models and aspects of the notional machine should be visualized to best allow students to learn about programming.

2.2 Fragile Knowledge in Programming

An understanding composed of *fragile* knowledge is one that may apply without problem in some cases but does not generalize well. It appears satisfactory during the learning process and is to some degree accepted by the learner, but will inevitably break down or make incorrect predictions in future scenarios that are sensitive to the particular flaws it has. Both misconceptions and mistakes can be related to *fragility* of the knowledge encoded in an individual’s mental model of the concepts at hand. From the constructivist viewpoint, a misconception is seen as a point at which an individual’s mental model makes a certain set of deviant predictions from those accepted by the larger academic community. Mistakes, on the other hand, are more transient errors that occur during the application of an otherwise viable mental model in a particular situation.

Perkins and Martin [51] discuss the role of fragile knowledge in programming in depth. They describe several different types of fragility that play a part in students’ struggles with writing correct code. Students may fail to utilize knowledge either because it is *missing* (i.e. never learned or forgotten) or because it is *inert* (i.e. not retrieved and applied to the task at hand). A student’s mental model may falter if knowledge is applied inappropriately, either by being *misplaced* (i.e. applied in the wrong context) or *conglomerated* (i.e. combined incompatibly with other knowledge). These specific kinds of fragility can be generally seen either as misconceptions or mistakes, but individual bugs may also involve aspects of several different types of fragility.

2.2.1 Misconceptions and Mistakes

In programming contexts, even a very small misconception can have immediate, catastrophic consequences. The true, internal computational models students attempt to learn are often strictly defined such that any bug at all can lead to incorrect results. While this is the way deterministic computation has to work, it can often be discouraging to the novice programmer. Perhaps even

more unfortunately, there are also cases in which a deviation from the correct computational model may be innocuous in all but a few cases – but in those few, can cause a program to fail completely. For code reading purposes as well, programmers need understand the details of the notional machine and work very precisely in order to correctly predict the behavior of a particular piece of code.

For example, consider a student whose model does not correctly account for the subtle differences between passing parameters by value and by reference. The student's model will be able to handle functions without incident unless the function modifies the value of one of the parameters and the argument passed to the function is later read. When this happens, the bug doesn't even manifest as a parameter passing issue - the apparent effect at first glance is that an unrelated variable has the wrong value. In fact, it is not immediately clear how one could properly trace the bug back to its source without referring to a conceptual model of the notional machine to reason backward through the program.

Simple misconceptions, such as those concerning individual language constructs, are not the only possible flaw in a mental model of the notional machine. If students are only exposed to each language construct in isolation, the result will likely be a fragmented understanding that will be fragile when applied to real programming problems. Even if a student possesses a correct understanding of each construct's semantics, it may not allow them to effectively write a program or check its correctness once written (e.g. by mentally tracing its execution). To avoid these sorts of misconceptions, each language concept should be presented as a part of a larger picture - a single model of the notional machine that describes how programs execute in a language. The notional machine provides a holistic context for individual constructs and makes it more natural to combine them together into complex programs.

Even if students have a correct understanding of programming concepts, they may still make mistakes when writing programs. Rather than a systematic flaw in a student's knowledge, a mistake may just be a result of not enough practice. Students who are learning to program are not just collecting a set of rules for how the notional machine, but they are also developing skills for reading and writing programs, which are realized bit by bit rather than all at once. For example, a student might understand the semantic difference between pass-by-value and pass-by-reference parameters, but still not have fully developed the intuition and skills involved in consistently using them correctly in practice.

2.2.2 Bugs and Debugging

Bugs are an expression of students' fragile knowledge in programming. The essence of debugging is usually thought of as the process through which one identifies undesirable behavior in a program,

tracks down the source of the problem, and fixes the issue. In an educational setting, it is also important to emphasize debugging as learning process. If the bug arose out of some misconception in a student's programming knowledge, identifying the misconception as the root cause of the bug and correcting the students understanding is crucial. That is, debugging an actual program is only useful insofar as it supports "cognitive debugging" via the location and elimination of misconceptions in the learner's mental model. It is not enough that the student's program becomes bug free, because the end goal is the content and viability of the students mental model - not the content of their program! Of course, the former entails the latter since the quality of students programs follows from the quality of their understanding, but the converse is not true. McCauley et al. provide a review of the literature on debugging and discuss implications for pedagogy and future research [52].

Programming bugs can be categorized by the types of errors in which they result. *Syntax errors* occur when the rules governing the set of symbols and/or acceptable structure for a program are violated. *Semantic errors* occur when a syntactically correct program is unable to produce a meaningful result because the semantic rules of the language do not define such a result (e.g. division by zero). These errors may be detected either at compile- or run-time, but are always conspicuous to the user (that is, it is obvious an error has occurred, but often not what is actually the root cause of the error!). A major class of semantic errors are *type errors* in which an operation is attempted with values of a type for which it is not defined. Different languages make various guarantees of *type safety*, most commonly in that they preclude certain kinds of type errors from happening in a program that successfully compiles in accordance with that language's type system. *Logical errors* occur when a program runs without producing a conspicuous error (i.e. no error message or program crash) but produces an incorrect result, which may or may not be obvious.

The manner in which tools designed to aid debugging inform a user of a possible bug can have significant implications for the productivity and academic growth of programming students. An empirical comparison by Robertson et al. [53] show that *negotiated-style* interruptions, in which a user is informed of a possible error but not forced to handle it right away, are superior for debugging contexts compared to *immediate-style* interruptions that require a user to address an issue before moving on to anything else. In particular, users who were provided negotiated-style interruptions attained better comprehension, showed increased productivity, and were reasonably effective at determining when a program was bug free (the immediate-style group was not). Robertson et al. postulate the immediate-style interruptions had strained users' short term memories. This is detrimental in its own right, but also may deter users from employing debugging strategies with high short-term memory requirements. As such, they believe immediate-style interruptions "promote over-reliance on local, shallow, problem-solving strategies. Other work also suggests attention and short-term memory play a crucial role in debugging processes [40].

2.2.3 Abductive Learning

The use of abductive reasoning may also play a role in the fragility of students understanding of programming. Abductive reasoning, as formulated by C.S. Peirce [54], is the process of adopting hypotheses that, when combined with previous knowledge, are apparently sufficient to explain observed phenomena, even if they are not necessary to do so (i.e. other, perhaps better, explanations may exist). Abductive reasoning can be used to very quickly generate explanations of novel phenomena, but at the cost of robustness because the explanations are ad-hoc adaptations of previous knowledge that often do not hold up in the general case.

Students often use abductive reasoning when they are learning to program. They interpret programs in accordance with their previous knowledge of the natural world or everyday life. This can be a potent tool for the beginning programmer - natural assumptions about the notional machine are easy to make and take little effort, and many turn out to be at least mostly correct. If students were to stop and verify every assumption they made, learning would progress at a much slower pace.

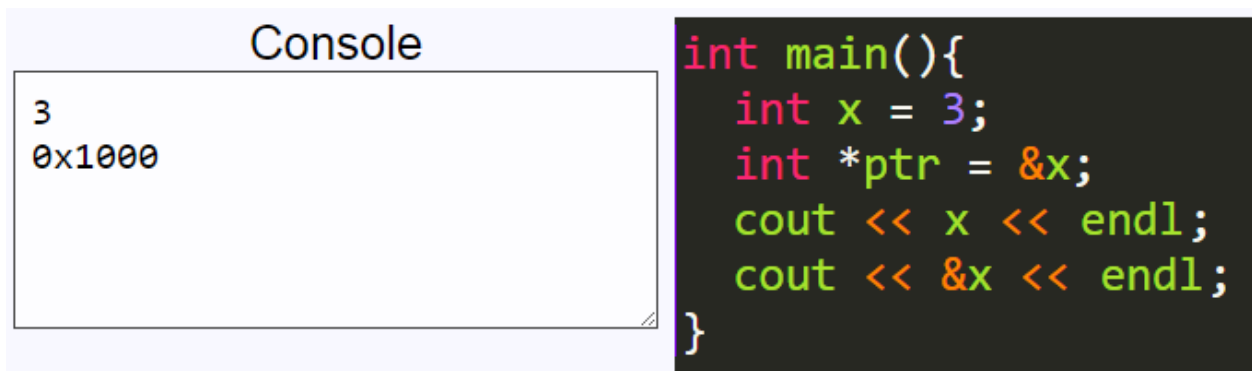
However, abductive learning can potentially lead to fragile knowledge, especially for learning activities that only give an indirect view of the subject matter. Learners may attribute meaning or significance to superficial similarities between specific observations and unrelated knowledge they already possess for understanding other domains. In essence, their explanations may be overfit to a small amount of information. Furthermore, learners often fail to detect when there are flaws in their explanations and generally do not perform thorough testing of the assumptions they make, if they are aware they are making them at all. These problems are compounded by the tendency of abductive explanations to override or inhibit learning from other means, such as formal, written descriptions of the way systems work. [55]

Taylor studied students learning to program in Prolog [56] and their use of abductive reasoning. In order to be effective Prolog programmers, students must have an understanding of both the declarative and procedural nature of the language. Based on observations of students, Taylor contends they do not have these understandings - rather, they attempt to adapt previous ways of understanding the real world and discourse between humans to account for the behavior of Prolog programs. This approach can be helpful and is sufficient for some problems, but is ultimately fragile when the deviation of Prolog's formal behavior from students intuitive assumptions becomes significant.

The “superbug” described by Pea [57] - an assumption that the computer is able to intuitively understand their intent - can be seen as a specific example of abductive reasoning gone awry. The superbug may arise from learners' application of strategies for participating in and processing human discourse to writing programs. The majority of everyday communication in which humans engage allows for details to be left to tacit interpretation and passes through a filter of common

sense. However, because programs are written in formal languages, the computer does not have any capacity to interpret any part of an algorithm that is not explicitly specified or to question a specification that seems unlikely to be what the programmer wanted (or to even know what that is!). Unlike another human in a conversation, the computer cannot ask for clarification if one of the previous situations occurs.

Thinking and working in the space of source code, which offers only an indirect view of what programs actually do, negatively affects the quality of abductive learning. Patterns in source code do not always translate to patterns in run-time behavior (and vice-versa), and an explanation formed to explain apparent phenomena in source code may be less powerful or even incoherent in terms of actual behavior at runtime. Superficial features in source code may be granted inappropriate or misplaced significance [12]. Problems with fragility in learned explanations are also exacerbated, since the explanations are not built on direct experiences with the notional machine, which the artifact students are really trying to explain in the first place.



```
int main(){
    int x = 3;
    int *ptr = &x;
    cout << x << endl;
    cout << &x << endl;
}
```

Console

3
0x1000

Figure 2.1: An brief example showing the use of the & operator. Because too little information is given, abductive learning from this example may lead to a fragile understanding.

As an example, consider the case of learning how to use the indirection (*) and address-of (&) operators in C++, a topic with which students often struggle. The code shown in figure 2.1 gives a short example illustrating the use of & to print the address of a variable. One explanation students could form in this case would be something along the lines of “use & whenever I want an address value”. This is sufficient to explain the situation at hand, and actually works fairly often, but is certainly fragile in the general case. If a separate pointer that holds an address as its value, applying the & operator gets the address of the pointer itself rather than the address it holds.

One approach to improving misconceptions is to provide more examples for students to learn from. Examples can also be strategically selected by an instructor to cover tricky cases or to provide counterexamples against faulty explanations students are prone to adopt. Unfortunately, this is not a panacea. If the additional information is given as several separate examples, students

may begin to develop non-viable explanations before seeing all the evidence and are faced with the challenge of reconciling piecemeal explanations that may not work well together. If more complex examples are used, students may either suffer from cognitive overload or resort to more complicated explanations than necessary to account for what they are seeing. For example, it may be difficult for beginning programmers to identify a simple, yet consistent explanation for all the uses of `*` and `&` from just the source code shown in Figure 2.2, even though one exists. (For now, let us assume students do not have access to the visualization of program state on the left side of the figure.)

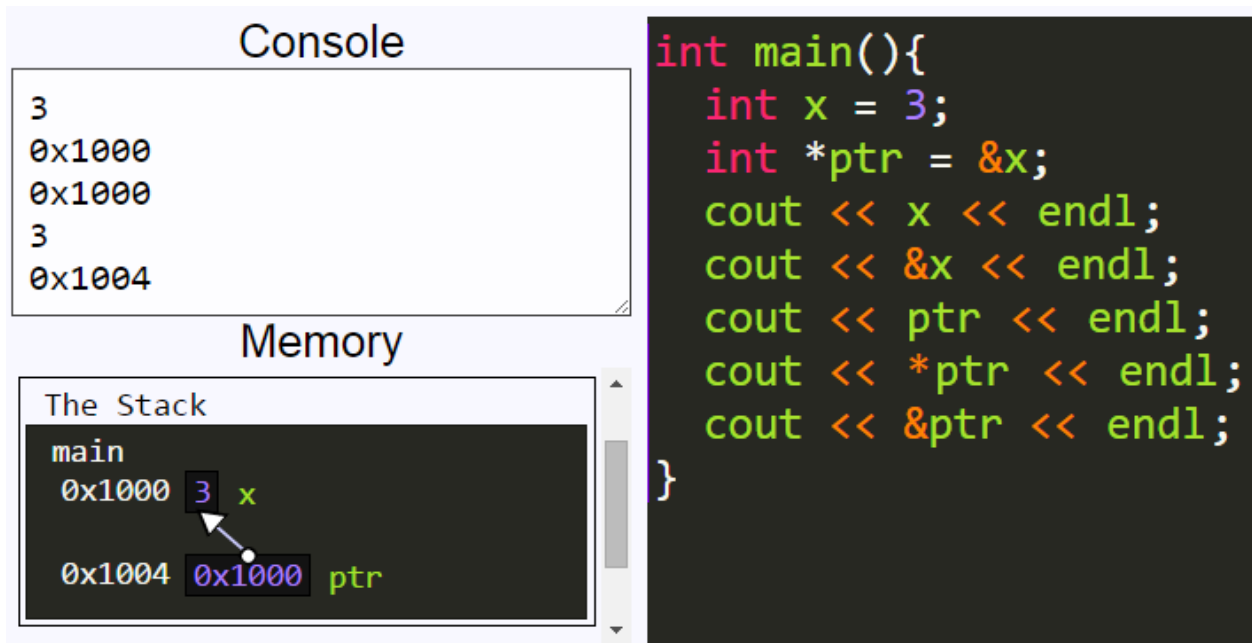


Figure 2.2: A more comprehensive example of the `*` and `&` operators. A visualization of the program state can alleviate problems of cognitive overload and provides a context in which the simple, correct explanations of the operators are naturally evident.

Ultimately, the goal is for students to learn what the `*` and `&` operators do in terms of program behavior. In this context, the rules are actually quite simple (e.g. `*` looks up the object at a given address and `&` yields the address of a given object). An understanding that encodes these rules is sufficient to explain any use of these operators in source code. However, abductively learning from examples of source code is unlikely to produce such an understanding due to problems of fragility, cognitive overload, and overly complicated explanations. Of course, instructors can stress the correct rules verbally, but they may still be competing with fragile, abductively learned explanations.

A strength of program visualization is allowing students to learn from experiences directly with a cognitive model of the notional machine. The visualization shown on the left side of Figure 2.2

depicts the program state implied by the given source code. The concept of objects (i.e. variables) that have both values and addresses is naturally represented and the strength of human visual processing is leveraged to provide complete information about the program state while still avoiding cognitive overload. While the source code in Figure 2.2 is complicated, the visualized program state is not, and learners need not turn to overly complicated explanations.

2.3 Tools For Learning Programming

This section provides a basic introduction to the varieties of visualization tools used to help students learn about programming. The goal of this section is to provide context for program visualization among other uses of visualization, but does not itself comprehensively cover the literature.

2.3.1 Program Visualization

Price et al. [17] define *software visualization* as "the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software." The focus of this work falls more specifically into the context of *Program Visualization*, particularly to the extent that the visualization enables the understanding of the notional machine in an educational context, but much that has been learned from previous work in other areas of software visualization is applicable for our purposes as well.

Figure 2.3 shows a diagrammatic view of different kinds of software visualization. Algorithm visualization is the largest subcategory shown, and includes systems for program visualization and also specialized systems for algorithm visualization (e.g. a system that animates sorting algorithms). Program visualization systems, on the other hand, are usually more generic and visualize the execution of programs at a lower-level. While a program visualization system can in theory be used for algorithm visualization (and thus is shown inside that category) by visualizing an implementation of the algorithm, this will be a less effective approach where the important aspects of an algorithm may be lost in the details of its implementation. This motivates the arrangement of our classification to include program visualization inside algorithm visualization, but to use a separate category for specialized algorithm visualization systems (in the literature, this is often just referred to as the algorithm visualization category).

We see program visualization as the category containing those systems which in some way attempt to illustrate the workings of the notional machine for users of the system. This often occurs in an educational context, and such is our focus. Within program visualization, there are a wealth of different aspects of the notional machine that may be visualized. Systems that support

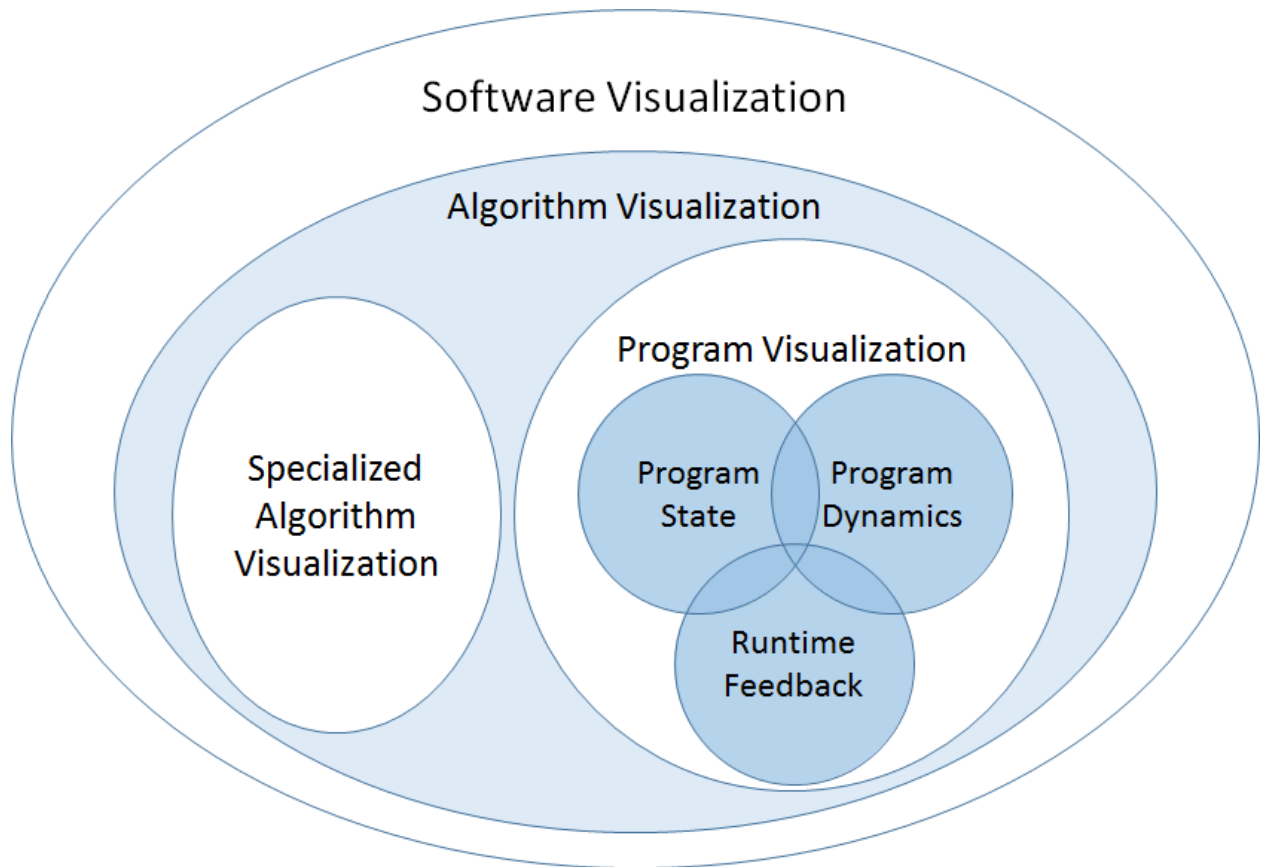


Figure 2.3: General categories of software visualization systems used to help improve *understanding*. Systems that use visualization for the *specification* of programs or software (e.g. visual programming) are not shown. Systems within the program visualization category can be seen as those that attempt to illuminate the workings of the notional machine. The fact that systems may support many qualitatively distinct ways to do this is represented by the subcategories within program visualization, but they are not necessarily exhaustive. Means of interaction or levels of engagement are orthogonal to the categorization shown in this diagram and are not represented here.

different subsets of these can be considered qualitatively different from each other. Some examples are shown in figure 2.3 and discussed here, but these are not exhaustive. Visualizing program state might include an indication of which statement is currently executing and what is contained in memory. To visualize program dynamics, on the other hand, a system would need to illustrate how the transition from one state to another occurs and help the user build an understanding of programs as dynamic entities. Providing additional runtime feedback to explain the effects of a particular statement or why an error has occurred provides a different view.

Another taxonomy, proposed by Oudshoorn et al. [58], provides a framework for describing *what* entities from the execution of a program need to be represented (and eventually displayed to the user), based on the characteristics of the target system. The first branch of the taxonomy considers the hardware architecture of the target system, and splits into uni-processor, distributed, and parallel systems. The second branch provides a framework for situating the software side of the target system, and first divides concerns based on whether the operating system, the programming language, the application, or some combination of these is the primary concern of the visualization. Furthermore, the language level is split into imperative, functional, logical, object-oriented, and parallel language paradigms.

With respect to this taxonomy, our primary focus in the present work is in the programming language category. In this context, the notional machine is key - the question of “what” needs to be represented that Oudshoorn et. al emphasize is essentially the question of which elements of the notional machine need to be shown to the learner to illuminate its inner workings. Our work is situated in a uni-processor environment, and while learning concurrent programming has several unique challenges, a working knowledge of the notional machine for such environments is still key in multi-processor environments. Likewise, illuminating the notional machine as a goal transcends the language paradigm, but different techniques are effective for different paradigms and the particular kinds of notional machines that exist for each.

We now briefly discuss algorithm visualization, visual programming, and visual debuggers for completeness and to contrast against program visualization, the area within software visualization on which the present work focuses. We also discuss the limited way in which visual debuggers support program visualization and motivate the need for a more complete picture of the notional machine.

2.3.2 Algorithm Visualization

Our work primarily concerns program visualization rather than algorithm visualization. In program visualization, the goal is to illuminate the notional machine behind the execution of a program. This both allows students to better understand language constructs themselves, but also how

to employ them in the composition of programs to solve problems. Algorithm visualization, on the other hand, focuses primarily on illustrating the workings of high-level algorithms that have already been designed to solve a particular task. Program visualization will be more closely tied to concrete program code, whereas algorithm visualization may be paired with pseudocode or abstract descriptions.

In the literature, program visualization and algorithm visualization are often discussed as two separate groups (see e.g. [17, 18]). In figure 2.3 we show algorithm visualization in a general sense, with program visualization as a sub-category. Technically, any program implements some algorithm, and for simple algorithms program visualization works well (e.g. finding the maximum element in a container). However, visualizing a code-based implementation of an algorithm is often too fine-grained to be a viable approach, and so we distinguish specialized algorithm visualization systems as those that are specifically designed to visualize algorithms in a manner appropriate for illustrating their behavior at a higher, conceptual level level. Because this is the sense in which algorithm visualization systems are discussed in the literature, unless otherwise specified we will be referring to these systems when using the term “algorithm visualization”. A summary of algorithm visualization research can be found in [59].

In program visualization, differences along the content ownership dimension in the engagement taxonomy become more pronounced than in algorithm visualization. Courses on algorithms and their analysis often focus on the presentation of an established set of important algorithms students should understand. When students are working with a visualization, it is unlikely that it would be helpful to change the program that is executing, because it would no longer implement the algorithm they are trying to learn about. In a sense, students are trying to learn what a particular program (i.e. algorithm) does, not how programming works. This is in contrast to a programming courses where students first learn the basic tools at their disposal and employ them in the construction of their own programs of increasing complexity. Students in these courses are primarily concerned with learning to understand how programming works, including both program comprehension and composition.

In a related sense, *effortlessness* [22] of visualization takes on greater importance in program visualization. Making even a small change to a program can drastically alter its execution and the visualization that needs to be presented. Each student should be allowed to explore the space of possible programs to their own desire, and any significant time investment needed to produce the visualization can be prohibitive. Allowing engagement with a student’s own input cases is often sufficient for an algorithm visualization where the algorithm (i.e. the “program”) is essentially fixed. The “what-if” questions that students might ask in the case of algorithm visualization are most often about how an algorithm handles patterns in different data, whereas in program visualization they may often involve potential changes in the code itself. As such, an algorithm

visualization system that allows students to change input data without unnecessary overhead could well be considered effortless, but an effortless program visualization system must allow students to readily change the code as well.

2.3.3 Visual Programming

As described in [17], visual programming deals primarily with the *specification* of programs using visual metaphors (e.g. “code blocks”). An advantage of visual programming systems in educational settings are that they allow learners to focus on the concepts behind semantic constructs without having to wrestle with the complex syntax of a textual programming language.

However, the primary means by which visual programming may enable students to better understand the notional machine is the extent to which the visual metaphors fit their intended conceptual behavior. That is, a visual programming language is an attempt to bring the language of program specification closer to the hypothetical language of the notional machine. Simply substituting visual metaphors into the representation of language constructs can be seen as essentially a way to preempt the process by which a student must develop their own mental model of the mapping from textual code to behavior running on a notional machine. While some metaphors may seem natural, this is an endeavor not necessarily supported by constructivist ideas or mental model theory.

Ultimately, a program written in a visual language is still an attempt to express a program through a static medium which cannot fully capture the sense of a dynamic, run-time entity. The best that visual metaphors can do capture the fullest sense of a programming cannot completely overcome the limitations of trying to express the fullest sense of a program as a dynamic entity at run time through a static medium. For example, once a programmer has become familiar with the placement of the pre-step, condition, body, and post-step of a for loop in textual code and understands the semantic role of each, it is hard to argue any more “visual” representation can offer more insight into the notional machine’s behavior while executing the loop.

2.3.4 Visual Debuggers

Traditional debuggers, even those with visual representations of program state, are generally not well-suited to support the kinds of learning experiences that will give beginning students insight into the notional machine. The focus of a debugger is, not surprisingly, enabling a programmer to track down bugs in their code. In this light, most debuggers are well suited for use by experienced programmers who already possess a working mental model of the notional machine. They can compare and contrast the information about a program’s execution shown by a debugger with their own mental trace of what a program should be doing and identify where the actual behavior

deviates from the expected. This sort of activity relies only on a visualization of the program's current state (see Figure 2.3) because the expert's mental model already accounts for and can fill in the missing aspects of the notional machine.

On the other hand, a student who is just learning to program does not already possess a viable mental model of the notional machine. They may not be well able to make predictions about a program's behavior and make meaningful inferences from whether and where the debugging output fails to match their assumptions. A beginning programmer in this situation is not the intended user considered when pragmatic trade offs are made in the design of debuggers to be used with for large and/or complex programs in practice. This often means debuggers favor coarse-grained execution and high levels of abstraction that are well-suited for expert users but not for beginners. If a debugger only supports coarse-grained (e.g. statement level) stepping, novices may not be able to fill in the gaps to understand what happened. Debuggers generally do not provide illustrations of the program dynamics involved in transitioning from one state to the next, and essentially lack the *continuity* principle.

A program visualization tool intended for use in education should take a different approach. The goal is not to enable students to track down a bug in a particular program (although visualization tools may certainly be helpful in that endeavor). Rather, the goal is to allow students to look into the black-box on which their programs run so that they can begin to construct an understanding of the notional machine in the first place. In particular, the visualization must be designed to illuminate what the notional machine is doing, and not just the current program state.

A debugger offers at the very least a step up from plain compilation and execution of a program - in that case the only feedback a student receives is in the form of program output. However, debuggers are not designed with education in mind and may not offer additional benefits beyond manual code traing. For example, students using the BlueJ visual debugger for course exercises showed no significant difference in learning than those who manually traced through program execution to complete the same exercises. [60]

2.4 Interaction and Engagement with Visualizations

Perhaps the single most discussed topic in the algorithm and program visualization literature is the importance of active engagement in learners' experiences with visualizations. As expressed in the influential report of a working group at ITiCSE 2002 [16], "...visualization technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity." This idea is rooted in the theories of constructivism and active learning, and many attempts have been made to analyze engagement specifically in the context of educational visualization. In this section, we discuss the origins of the engagement hypothesis, the development

of taxonomies to categorize different kinds of engagement, and the state of supporting evidence from experiments in the literature. We also discuss limitations of the engagement hypothesis and taxonomies, and the potentially increased significance of factors beyond engagement in the specific context of program visualization as opposed to algorithm visualization.

2.4.1 The Engagement Hypothesis and Taxonomies

A 2002 meta-study [48] by Hundhausen et al. brought much attention to the importance of learner engagement in algorithm visualization. The authors looked at between-subjects experimental studies of the effectiveness of various algorithm visualization systems and techniques. The results of the evaluations included in the meta-study varied widely, with some interventions having no statistically significant impact, but the level of learner engagement emerged as the best predictor of whether significant results were achieved. Those visualizations that actively engaged students were more likely to produce significant, positive results. The authors of the meta-study argued that the evidence indicated “*how* students use AV technology has a greater impact on effectiveness than *what* AV technology shows them.”

Drawing on support from this meta-study, a working group at ITiCSE 2002 [16] argued that no matter how well visualizations are designed, they are of little use unless they engage learners in active learning. In order to establish a common language for characterizing the kinds of engagement enabled by visualizations and to provide a framework for experimental studies of the role of engagement in visualization effectiveness, the members of the group proposed a taxonomy of learner engagement with visualization technology. In the years since, a number of revisions to the original taxonomy have been proposed (e.g. [19, 20, 18]), and several experimental studies have been carried out to compare the effectiveness of learning activities situated at different levels of engagement. The engagement taxonomy, its revisions, experimental studies of engagement, and the implications of each are discussed below.

2.4.1.1 The Original Engagement Taxonomy

The original taxonomy [16] included six different forms of learner engagement, roughly ordered in terms of increasing engagement as *no viewing*, *viewing*, *responding*, *changing*, *constructing*, and *presenting* (see Figure 2.4). However, the authors make clear that the categories of the taxonomy are not mutually exclusive - many forms of engagement may occur together in one learning activity. The *no viewing* category encompasses any activity that does not involve an educational visualization, and likewise the *viewing* category simply amounts to some kind of work with a visualization and is implied by any of the other forms of engagement. Beyond the *no viewing* and *viewing* levels, the taxonomy was not intended to be strictly ordinal, although the original authors believed the

higher categories to be roughly in increasing order of engagement.

Mode of Engagement	Description
No Viewing	There is no visualization to be viewed.
Viewing	The visualization is only looked at without any other form of engagement.
Responding	Learners are presented with questions related to the visualization.
Changing	Modification of the visualization is allowed, for example, by varying the input data set.
Constructing	Learners are expected to create their own visualization of a program or an algorithm.
Presenting	Learners present visualizations to others for feedback and discussion.

Figure 2.4: The Original Engagement Taxonomy. Reproduced from Myller et al. [20].

2.4.1.2 Revisions and Additions to the Engagement Taxonomy

Since the creation of the original engagement taxonomy, several revisions and additions have been proposed. Lauer [19] drew attention to issues of heterogeneity in the *viewing* and *constructing* categories of engagement. In particular, the *viewing* level was originally defined broadly to contain both passive viewing activities as well as active viewing activities in which the learner controls the progression through the visualization. Additionally, Lauer points out the activities categorized as *constructing* in [61] involve students selecting fine grained operations that should be applied in a visualization to execute a particular algorithm, while in [62] students select the most relevant frames from a set of automatically generated snapshots of algorithm animation in order to compose a final visualization. The two are clearly different sorts of engagement. Indeed, in more recent taxonomies an *applying* category is added for activities in which students engage by playing the role of the computer and selecting the next action to be executed.

Myller et al. [20] proposed an extension to the original engagement taxonomy to include a distinction between *viewing* and *controlled viewing*, as well as additional categories for new modes of engagement they identified. The *reviewing* category encompasses activities in which an individual engages in critical, evaluative activities regarding the visualization or the content being visualized. The *entering input* and *modifying* categories were added to specifically account for instances in which the user of the system is allowed to change the inputs or source code of the visualization, respectively. These two categories are particularly relevant to program visualization in which the user may be allowed much greater control over the content being visualized (whereas a particular algorithm is usually chosen and fixed in algorithm visualization).

2.4.1.3 The 2DET and Content Ownership

The most recently proposed engagement taxonomy, the Two-Dimensional Engagement Taxonomy (2DET) [18], adds an orthogonal dimension for the level of content ownership implicit in the learning activity. The levels along this dimension, ordered by increasing degree of ownership, are *given content*, *own cases*, *modified content*, *own content*. The other dimension of the 2DET concerns the learner’s mode of direct engagement with a visualization and is based on earlier taxonomies [16, 19, 20]. The seven different categories along this dimension, in roughly increasing order of engagement, are *no viewing*, *viewing*, *controlled viewing*, *responding*, *applying*, *presenting*, and *creating*. The *reviewing* category from the extended engagement taxonomy of Myller et al. has been included in the *presenting* category because it arguably involves the same kind of engagement, regardless of whether the target audience is oneself (*reviewing*) or others (*presenting*). Figure 2.5 shows the categories of the 2DET as well as a brief description of each, and figure 2.6 shows the hypothesized trend of increasing engagement at higher levels of the taxonomy.

Table I. The Categories of the 2DET Engagement Taxonomy

The direct engagement dimension		
#	Level	Description
1	No viewing	The learner does not view a visualization at all.
2	Viewing	The learner views a visualization with little or no control over how he does it.
3	Controlled viewing	The learner controls how he views a visualization, either before or during the viewing, for example, by changing animation speed or choosing which images or visual elements to examine.
4	Responding	The learner responds to questions about the target software, either while or after viewing a visualization of it.
5	Applying	The learner makes use of or modifies given visual components to perform some task, for example, direct manipulation of the visualization’s components.
6	Presenting	The learner uses the visualization in order to present to others a detailed analysis or description of the visualization and/or the target software.
7	Creating	The learner creates a novel way to visualize the target software, for example, by drawing or programming or by combining given graphical primitives.
The content ownership dimension		
#	Level	Description
1	Given content	The learner studies given software whose behavior is predefined.
2	Own cases	The learner studies given software but defines its input or other parameters either before or during execution.
3	Modified content	The learner studies given software that they can modify or have already modified.
4	Own content	The learner studies software that they created themselves.

Figure 2.5: The categories of the direct engagement and content ownership dimensions of the 2DET.

The rationale given by Sorva et al. [18] for including a separate dimension for content ownership is in part that learners may be more invested in and motivated by content they have had some hand in creating, but their formulation also provides clarity lacking in some parts of earlier taxonomies. Any learning activity with a visualization naturally includes some degree of content ownership and a one dimensional taxonomy cannot accommodate this well. For example, the original engagement taxonomy [16] did not thoroughly account for content ownership and thus

whether students view a visualization of given code, code they are allowed to change, or entirely their own code (potentially very different learning activities!), the level of engagement would just be *viewing*.

The extended engagement taxonomy of Myller et al. [20] had earlier attempted to address content ownership by adding the *entering input* and *modifying* categories, but these are awkward to consider as a full learning activity on their own - they must be paired with some other mode of engagement to make sense. For example, students might enter input and then engage in controlled viewing of the result, but it doesn't make sense to just enter input! This speaks to the need for an additional, orthogonal dimension so that one always considers a particular pairing of both direct engagement and content ownership to characterize a learning activity.

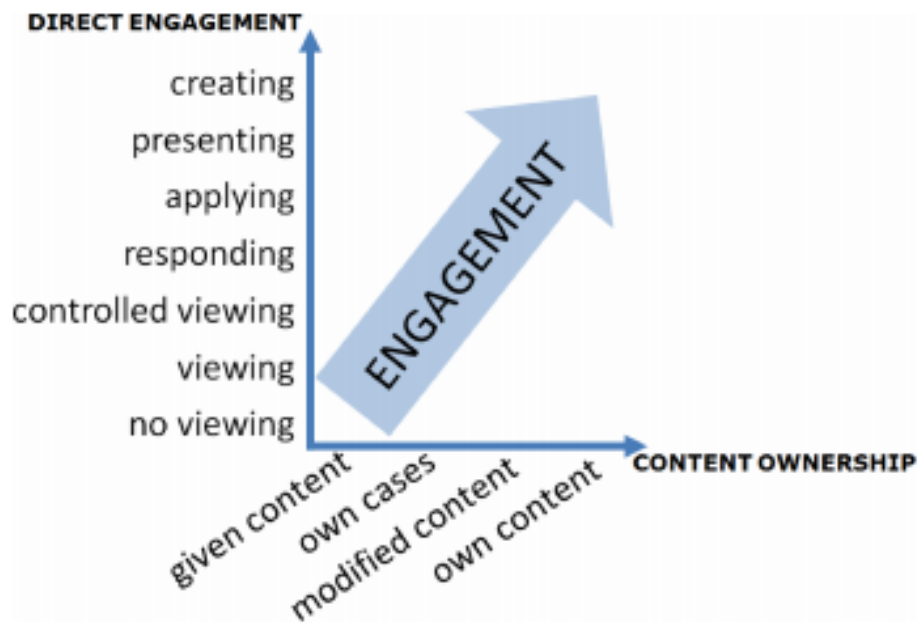


Figure 2.6: The categories of the 2DET, ordered according to the hypothesized trend of increasing engagement.

Sorva et al. [18] also acknowledge potential limitations of the 2DET (and engagement taxonomies in general). Fundamentally, giving students a task from a particular category of the taxonomy does not guarantee students' will truly participate at the desired level of engagement (see e.g. Grissom et al. section 4.5 [63]). The *responding* category is also problematic because different kinds of questions yield very different levels of engagement. That being said, well-designed questions that actually concern what is going on in the visualization can easily engage students in working toward high level learning objectives of Bloom's taxonomy, so the placement of the *responding* level in the 2DET seems appropriate.

2.4.1.4 Consumers and Producers

Similarly to the taxonomy of effortless visualization proposed by Ihantola et al. [22], we consider separately the roles of producer and consumer in interacting with visualizations. For example, a student who views a pre-made visualization and responds to pop-up questions is acting as a consumer, while a student who creates their own visualization as part of an assignment is acting as a producer. Instructors may also play a complimentary role, either by producing a visualization for students to view or evaluating one that students have made, but our focus is on the student's role. We can group the categories of direct engagement according to whether they fill the role of consumer or producer. The consumer role covers *applying*, *responding*, and *analyzing*, because these are all modes of interaction in which the learner is using the visualization and drawing knowledge and insight out of working with it. On the other hand, *constructing* and *presenting* are modes of interaction in which the student primarily plays the role of producer in that they are using their knowledge to create a visualization for consumption by others. Of course, as the level of content ownership increases, students can additionally act as producers of the code that is to be visualized.

2.4.2 Empirical Results in Learner Engagement

While the evidence for the importance of active engagement in general is incontrovertible, empirical support for different learning outcomes arising from different categories in the engagement taxonomy is incomplete. Experiments (see e.g. [21, 18]) have provided support for some the fundamental pieces of the engagement hypothesis - namely that viewing a visualization is better than not viewing one at all, and that it is important for students to actively control the visualization. Higher levels of engagement (e.g. responding, applying, constructing) have been found more effective in certain contexts, but there is not enough evidence to conclude they are categorically better. In this section, we discuss the state of the engagement hypothesis and taxonomy, as well as the relevant evidence from the literature.

2.4.2.1 No Viewing vs. Viewing vs. Higher Levels

The distinction with the strongest empirical evidence from the literature is between the *no viewing* level and others, but this only supports that some use of PAVs is better than none at all. Part of the lack of evidence for differences between other levels of the engagement taxonomy is due to a lack of experiments targeting those levels. In a 2009 survey of successful visualization systems [21], nearly three quarters of the experiments that compared multiple levels of the engagement hierarchy compared against a higher level against the *no viewing* level, while the remaining quarter compared higher levels against the *viewing* level. There were no comparisons of two levels higher than *viewing*.

An issue with interpreting results concerning learner engagement is that the transition from the *no viewing* level to others (often to *viewing*) is difficult to consistently frame throughout the literature. For example, several experiments in Hansen et al. [64] compare students who used the HalVis visualization to those who did not. In the Hundhausen et al. meta-study [48], these experiments are placed into both the cognitive constructivism and epistemic fidelity categories. In the former case the difference between the student groups is considered as a change in the level of engagement, but in the latter it is considered as a change in the learning medium.

Lauer performed an empirical comparison [61] of the effectiveness of the *controlled viewing*, *changing*, and *applying* modes of engagement for students learning about the Fibonacci heap data structure, which was introduced to the students during lectures before the experiment. No significant difference was found between the groups, but the author suggests the lectures may have had a large enough effect on the students learning that the extra activities working with the visualization may have produced an effect that was relatively too small to be detected.

Grissom et al. [63] investigated the use visualizations of simple sorting algorithms at different levels of engagement. They found that students working at the *responding* level had greater increases in performance than those working at the *uncontrolled viewing* level, although the result was not statistically significant. Both approaches were superior to not working with a visualization at all, and the result was statistically significant for *responding*.

2.4.2.2 The Constructing Category

Hundhausen and Douglas [65] conducted an experiment in which students who had previously learned the QuickSelect algorithm either interactively viewed a visualization of the algorithm or constructed their own using traditional media (i.e. paper, art supplies, etc.). Unique to this study is that the visualizations (both the one supplied to the first group and those created by the second group) were designed using story content. The experiment essentially compared students working in the *controlled viewing/down cases* category with the *constructing/down cases* category, and found no statistically significant difference between the two.

Hundhausen and Brown [66] found that students using the ALVIS Live! system to develop algorithms and construct visualizations outperformed those who had used a traditional text editor and presentation media. Hbscher-Younger and Narayanan [67] investigated algorithm visualization using traditional media and found students who had created visualizations (i.e. worked at the *constructing* level) learned more than those who had viewed and evaluated those visualizations. However, students who created visualizations were selected on a volunteer basis and thus may have represented a more motivated group than those who only viewed.

An empirical evaluation of WinHIPE [62] compared learners using the system to build visualizations of a tree traversal algorithm (i.e. selecting which frames to include in a final animation)

to those only viewing animations that had previously been generated by others. Students who had engaged in *constructing* visualizations spent significantly more time working with the system and had improved learning outcomes compared to those who viewed animations. An important consideration is that in this study, the *constructing* task only involved choosing which of several automatically generated frames to include in an animation. This is only a small set of activities normally included at the constructing level of engagement, and it may be that the careful consideration of the algorithm required as part of constructing a visualization can be distilled as the most impactful element of construction activities.

This experiment should also be understood in the context of effortless algorithm animation - the primary work of students in the builder group was to determine which frames to include in the animation. This is only a small set of activities normally included at the constructing level of engagement, and the rest were essentially made "effortless" for students, so the results may not extend to other forms of constructing. It may be that the careful consideration of the algorithm required to determine which frames are relevant can be distilled as one of the most impactful elements of construction activities.

2.4.3 Limitations of the Engagement Hypothesis and Taxonomies

In this section, we discuss concerns relating to the origin and interpretation of the engagement hypothesis and taxonomies. While it is clear that engagement is important in a general sense, it may not be appropriate to treat it as the primary factor in the effectiveness of a visualization.

2.4.3.1 Origins of the Hypothesis

The authors of the initial meta-study [48] looked at between-subjects experimental studies of the effectiveness of various algorithm visualization systems and techniques. The results of the evaluations included in the meta-study varied widely, with some interventions having no statistically significant impact, but the level of learner engagement emerged as the best predictor of whether significant results were achieved. Those visualizations that actively engaged students were more likely to produce significant, positive results. The authors of the meta-study argued that the evidence indicated "*how* students use AV technology has a greater impact on effectiveness than *what* AV technology shows them."

The authors grouped together the experiments surveyed according to broad categories of theoretical underpinnings that could be presumed from the independent variables used by each and the ways in which those variables were manipulated. Of the theories identified, the two predominant groups were *Epistemic Fidelity* (i.e. "what") and *Cognitive Constructivism* (i.e. "how"). The Epistemic Fidelity theory is based on the idea that graphics and animation have the power to represent

a conceptual model of the material students are learning, and the fidelity of the representation used in the visualization is a major factor in the quality of learning. The Cognitive Constructivism theory centers around individuals actively constructing knowledge by integrating new experiences into their existing knowledge base. According to this theory, the higher the level of engagement afforded by the visualization, the higher the quality of learning.

Study	Independent Variable	Significant difference
Price [68]	Animated vs. Non-Animated Debugger	No
Lawrence Chapter 4.4 [69]	Data set size Data representation style	No
Lawrence Chapter 7 [69]	Representation Color	Yes
Lawrence Chapter 8 [69]	Order of Medium Order of algorithm presentation	No
Gurka [70]	Learning medium (animation vs. no animation)	No
Mulholland [71]	Tracing medium (textual vs. graphical)	Yes
Hansen et al. Study III [64]	Learning medium (HalVis exercises vs. pen/paper exercises)	No
Hansen et al. Study VI [64]	HalVis features	No
Hansen et al. Study VII [64]	HalVis views	Yes
Hansen et al. Study VIII [64]	HalVis views	Yes

Figure 2.7: Studies that were grouped into the *Epistemic Fidelity* category in the Hundhausen et al. meta-study.

The authors then compared the proportion of studies within each group that yielded a significant result. The *Cognitive Constructivism* category was found to have a higher proportion than *Epistemic Fidelity*, and this was interpreted as evidence that “how” students use visualizations has a greater impact than “what” students see. However, it is not surprising that changes in the level of engagement would more frequently produce significant results, since the level of engagement can more cleanly framed as a matter of degree.

This is not the case for the “what” question. Essentially, the “what” variable has much higher dimensionality, and we should not expect to find significant differences along each possible dimension that would could change the representation that is being visualized. For example, consider the studies Hundhausen et al. identified as testing the Epistemic Fidelity theory, shown in Figure 2.7. The work of Hansen et al. [64] found a significant difference in learning outcomes between different conceptual views of an algorithm visualization, but the majority of studies found no significant

difference. Should we say that “what” we show students only matters a little bit because overall most changes to “what” that happened to be studied produced no effect? No! We should say that some parts of the “what” appear to matter a lot (e.g. conceptual views), even though other parts may not be influential.

The point is that while the *Cognitive Constructivism* theory was as a whole best able to predict whether studies would find a significant impact on students, this does not necessarily mean it is wise to focus on this area above others. The other categories considered by the authors of the initial meta-study were either not well studied enough to be included or not well suited to be judged as uniform category. In particular, the studies that were grouped into the *Epistemic Fidelity* category vary widely, and should not be considered as testing the same variable. A strength of the engagement taxonomy is that it makes it easy to frame studies comparing different levels of engagement, but researchers must be careful to consider engagement as only one (very important!) piece of the bigger puzzle.

2.4.3.2 A Bigger Picture

A cursory reading of the engagement hypothesis may lead one to discount the potentially significant contributions of other factors to the effectiveness of program visualization. However, one must be careful to consider the specific context in which certain kinds of engagement occur, including the particular visualization system used, the educational setting, and even the students themselves. It is easy to imagine that a particular kind of engagement might work better than another in a specific context, but this does not necessarily generalize to all. The level of engagement is certainly not sufficient to single-handedly account for the success or failure of a particular approach.

With respect to the taxonomy itself, it is difficult to categorically establish that one form of engagement is more effective in learning than another. The best that can be done is to collect together the results of several experiments all carried out on different visualization systems, but unless one assumes the level of engagement is independent of all other factors, it is difficult to identify it as the sole influence on success or failure of particular systems. Some experiments may have been carried out with systems that just do not provide the adequate support for a particular mode of engagement, but that does not mean that mode of engagement is ineffective or that it is not possible for other systems to support it. Likewise, perhaps different modes of engagement are more productive for certain populations of students.

A number of factors may potentially limit the significance of experimental differences found between students working with different categories of direct engagement. Many studies are performed in a classroom setting, where the relative contribution of visualization exercises is small compared to that of reading a textbook, attending lectures, or completing homework assignments (cf. [61]). These other influences may “drown out” the effects of the experimental intervention. In

older versions of the engagement taxonomy, some categories were too vague and different visualizations in the same category showed conflicting results. For example, a single viewing category is insufficient when one considers the importance of whether the learner takes an active or passive role during the viewing [19].

Overall, in this dissertation we seek to shine a different light on approaches to program visualization to provide more perspective than just the level of engagement. While the evidence for active learning in general is unquestionable and certainly applies to program visualization, the level of engagement seems insufficient to single-handedly account for the success or failure of a particular approach. Instead, we propose that the ultimate goal of a program visualization system is to aid students in learning about the notional machine. This involves both “how” students interact with the notional machine and “what” the system shows them about the notional machine - the two cannot meaningfully be considered separately.

2.5 Effortless Visualization

One of the main obstacles to the adoption of visualization tools in mainstream education is the large amount of time that may be required to produce quality visualizations [16]. A software visualization system can be considered effortless if it minimizes the amount of up front time and work that must be invested before the visualization can be used. It is important to stress that an effortless approach to program visualization does not and should not attempt to remove the need for “effort” on the part of the learner. Active engagement, which is essential for high quality learning, involves a great deal of effort! Rather, the effortless approach is to eliminate unnecessary effort required to use a system, but not related to the learning process. Once this is accomplished, students can focus their effort on the most important aspects of the learning process.

An example, we can consider the approach taken in the WinHIPE visualization system [72] to constructing visualizations. The system automatically generates a sequence of static frames, each corresponding to a specific point in the evaluation of an expression. Because some evaluations may involve a very large number of individual steps, the user is allowed to manually select which frames should be included in a customized animation and can save the animation for later. In an experimental evaluation of the system [62], students who customized a visualization performed significantly better than those who had viewed visualizations created by others. The construction of visualizations was effortless for students in the sense that they did not have to laboriously create each frame, but they still invested meaningful effort in choosing which frames to include and making customizations. The careful consideration of the algorithm required to determine which frames are relevant can be distilled as the most impactful element of construction activities, and so should be the part where students focus their effort.

Ihantola et al. [22] describe a taxonomy of effortless creation of algorithm visualizations including dimensions for 1. the scope of content to which a visualization system is applicable, 2. the ease with which it can be integrated into an educational setting, and 3. the modes of interaction it supports. Although their original paper addresses algorithm visualization, the work is applicable to program visualization as well.

The scope dimension is split into four categories of increasing generality: lesson-specific, course-specific, domain-specific, and non-specific. While non-specific systems are capable of visualizing almost anything, a system in that category will not necessarily support all the more specific categories as well. For example, any tool that supports the creation of animations in general can be seen as a non-specific system, but if they do not make it particularly easy to create algorithm or program visualizations, they may fail to support any of the more specific levels.

The integrability dimension considers features a system may have that allow it to be easily integrated into teaching and learning activities. The authors give a non-exhaustive list of possible features including easy installation, customization, platform independence, internationalization, documentation, interactive prediction support, course management support, and integration of hypertext. A system with more of these features is considered more effortless, but the authors give no relative ranking of importance between them.

The interaction dimension of the taxonomy considers both producers of visualizations and the consumers who view them. Likewise, two different sorts of interaction are in view: producer-system interaction while the visualization is being created and visualization-consumer interaction after the visualization has been created. Producer-system interaction can be characterized by a set of use cases instructors may have for the visualization software and the corresponding level of effort needed to achieve each. Consumer-system interaction concerns the modes of engagement supported and is defined by categories modeled after the engagement taxonomies discussed above.

There is an intimate relationship between the effortlessness of a system's producer-system interaction and the ability for it to efficiently support higher levels of content ownership in the 2DET. If students must invest significantly more work in order to visualize their own content they may be less eager to do so and fall back to working with given content (e.g. examples prepared ahead of time by the instructor). Motivation issues aside, minimizing overhead work simply give students more time to explore more examples of their own construction.

Our focus on allowing students to work with their own code means that effortlessness is crucial in the interaction between students and the system. If students must invest significantly more work in order to visualize their own content they may be less eager to do so and fall back to working with given content (e.g. examples prepared ahead of time by the instructor). Thus, students should not have to do anything extra to the code they write in order to visualize it, such as adding special annotations or manually setting up parts of the visualization. This kind of work is inauthentic and has

little educational value, and students may see visualization activities as a waste of time, even if the visualization would turn out to be worth the effort. Motivation issues aside, minimizing overhead work simply gives students more time to explore more examples of their own construction.

2.6 Other Trends in Program Visualization

In 2009, Urquiza-Fuentes and Velzquez-Iturbide presented a survey of program and algorithm visualization systems (PAVs) that had been successfully evaluated. [21] The authors identify several trends in the field. Simply viewing animations can improve learning, and textual/narrative contents can be used to provide deeper explanations of the visualization or express feedback to students. When students construct their own visualizations or write accompanying descriptions, positive changes in attitudes toward the subject matter have been observed. Interfaces for students to construct their own visualizations tend to be carefully designed for students.

Another trend identified in the survey is that about half of the PAVs considered have only been subjected to usability tests, and many of these are not formal or rigorous evaluations. While usability is an important factor in the success of a system, it is not enough to justify using a system in an educational setting. Of the PAVs that have been evaluated for educational effectiveness, the authors find the script-based systems are useful for generating visualizations that can be made available to students, but less so for the students to use themselves. On the other hand, compiler-based systems tend to be easier for students to use at higher levels of engagement (e.g. changing, constructing, presenting). Some systems also attempt to find a middle ground, either by providing multiple interfaces for beginners/experts or by providing default behaviors with optional customizations for parts of visualization creation process.

Because the design of their survey was to only include evaluations that produced positive results, it is difficult to draw conclusions about the correlation between aspects of the program visualization systems considered and their effectiveness. Rather, the conclusions drawn should be seen as trends of the PAVs that are being used and evaluated and that prove successful. For example, even though 75% of the systems surveyed involved narrative and textual content, we cannot necessarily conclude they are an essential feature for a successful system. It may well be an equal proportion of systems that were ineffective also involved those kinds of content. However, one can conclude that the majority of successful systems do include narrative contents and that this is a common practice in the field. It has an intuitive appeal, and as long as the flow of information can be controlled and is presented in a way that does not overwhelm the learner, there is little reason to think adding explanations of what goes on in the visualization would be detrimental.

Urquiza-Fuentes and Velzquez-Iturbide also identify additional “features” that may play a role in the effectiveness of visualizations for educational purposes. These are:

- **Narrative contents and textual explanations** Written explanations accompany the visualization to provide a more thorough explanation of what is going on.
- **Feedback on students' actions** If students are given choices or asked to predict outcomes in the visualization, explicit feedback can be given as to why their action was correct or not.
- **Extra time using PAV** Visualization exercises often require students to spend additional time working with a topic.
- **Advanced features** Some visualizations include specialized or advanced displays or interfaces.

CHAPTER 3

Contemporary Program Visualization Systems

In this chapter, we give a brief description of several contemporary program visualization systems. The set included here is not intended to be exhaustive, but rather to give the reader an idea of the current state of the field. Many different varieties of visualization systems exist (see e.g. [17]), but we are primarily concerned with our work is focused on generic program visualization systems as described in Sorva’s review [18]. This kind of system can visualize most any code written in its target language and is able to give a holistic view of the notional machine as it dynamically executes program code. We also focus on systems that are highly interactive and support visualization of students own code. These systems fundamentally transform the experience students have when learning to write programs by providing an environment where running their code is a truly “hands on” experience with the notional machine.

3.1 Systems for Preparing Visualizations

A number of systems are designed specifically to support visualizations prepared ahead of time by one person, often by an instructor or expert, and viewed at a later time by learners. For the learners, these systems only support engagement with *given content*, because it must be prepared ahead of time, but may involve several modes of engagement. At the least, *viewing* would be supported, but may suffer from issues of learner passivity and thus *controlled viewing* is usually preferable. Systems may also support engagement via *responding* if there are facilities for the producer of the presentation to include questions with the visualization (e.g. “pop-up” questions at key points of the visualization). Finally, *applying* may be supported from a limited standpoint, but complex feedback beyond checking whether students perform the correct operation would require significantly more work on the part of whoever prepares the visualization.

Of course, if students themselves are given the task of using the system to create a visualization for others, then this would constitute *creating*, and if they are asked to explain a visualization or its content to others, then they are *presenting*. In the discussion below, we will not explicitly address

these possibilities unless a system provides specific features to support them.

The Javascript Algorithm Visualization Library (JSAV) [73] is an HTML5-based toolkit designed to support AV features that encourage students to engage in active learning. The library supports *controlled viewing* of animations as well as *responding* to pop-up questions during a visualization. Interactive proficiency exercises that allow students to simulate the algorithm on their own reflect the *applying* level. Students may also work with their *own cases* as well as *given content*. Of particular note is that JSAV’s support for procedural generation of diagrams/visualizations makes constructing or modifying them much more efficient (i.e. effortlessness), and also gives them a cohesive look and feel.

The OpenDSA “active-eBook” [74] (online at <http://algoviz.org/OpenDSA/>) goes beyond a regular hypertextbook by including closely integrated interactive visualizations/simulations enabled by the JSAV library, as well as interactive assessment activities. The development of OpenDSA has been open-source and community-based, in an effort to combat the large development cost of high-quality interactive elements. OpenDSA is one of the closest examples to the kind of IDEAL content we envision, but currently the authoring system behind it requires web programming skills and is specialized toward algorithm visualization. As described in more detail below, we would like to explore ways to support authoring interactive content in general (applicable to varied fields) without programming requirements.

Sirki presents a JavaScript library [75] for creating program visualizations that can be seamlessly integrated with other web content. The library works similarly to JSAV, but is focused on program visualization rather than algorithm visualization. The library provides an API for creating language-specific visualizations by selecting from common operations including fetching values, evaluating operators, and assigning values to variables. However, the library does not directly support complex or nuanced features of individual languages. It supports the *controlled viewing* level of engagement through built in functionality for navigating forward/backward step-by-step through a finished visualization as well as showing text alongside particular steps. Visualizations can be customized and may be styled using CSS.

3.2 Program Visualization Systems

Bradman [76] was a program visualization system whose design goal was to help novice programmers develop a concrete mental model of how program execution works. The creators refer to it as a “transparency debugger” because it allows beginning programmers to see an explicit representation of how their program works. Bradman accepts a syntactically correct C program as input and displays several different windows visualizing the code currently executing, the program’s variables, input/output, and most notably specific natural language explanations for the statement

currently executing. When surveyed, students who used Bradman with explanations were statistically significantly more likely to report it had helped them find a bug than those who used the same system but without explanations. [77]

The Jeliot program animation system allows visualization of Java programs. Students can write and visualize their own programs, and the system supports engagement at the controlled viewing level. Two major principles behind Jeliot's design are *completeness* and *continuity*. All aspects of an executing program are represented in the visualization, and the transitions between states are made explicit. This can be tied back to the idea of illuminating the notional machine - students will have a much harder time developing a viable mental model if important details are not visualized. Jeliot has been the subject of a number of empirical evaluations, and in particular has been shown to positively affect learning and attention. A review of research using the Jeliot environment can be found in [78].

The Online Python Tutor [79] is a web-based program visualization tool for the python language that allows students to write and visualize their own code. A major strength of the system is its ease of use and integrability into other educational material. Using the tool is as simple as visiting a webpage and visualizations can easily be embedded into other web content, such as web-based eTextbooks.

The source code for the program currently being visualized is shown, and both the previous and next line to be executed are indicated with a graphical arrow. The program is run in its entirety ahead of time in order to produce a line-by-line trace from which the visualization is generated. This means users are able to see the total number of steps in the execution of their program ahead of time and either move forward/backward one line at a time or drag a slider bar corresponding to the entire sequence of steps.

The state of the program is visualized as a downward growing sequence of stack frames, each corresponding to one function invocation in the current call stack trace. Each frame shows the function name, local variables, and arrows connecting variables to objects in a separate heap display. The heap display shows visual representations of objects that are currently being used by the program as well as arrows to indicate any references between them. Compound data types appropriate for introductory courses (e.g. lists, dictionaries, class-type objects) are rendered in an intuitive manner.

Beyond the obvious difference in target language, Labster and the Online Python Tutor differ in the way they present several aspects of the notional machine. Labster shows each function invocation as a separate entity with its own source code and current point of execution (in line with the dynamic-logical model of execution [80]), whereas the Online Python Tutor highlights the overall point of execution (i.e. previous/current line) on the single copy of the source code. Labster visualizes expression evaluation, while the Online Python Tutor only allows stepping line-

by-line. Additionally, transitions from one program state to the next are not animated in the Online Python Tutor, which may make it more difficult to follow complex operations that occur in one step. These differences allow Labster to achieve more completeness and continuity [81] in the way the notional machine is presented.

WinHIPE [72], an IDE for functional programming, supports interactive program tracing and animation. It adheres to the effortless paradigm of visualization in that the system automatically generates visualizations for written code and there is no overhead for the user to work with the visualizations. The goal of this approach is to provide high value visualizations at low cost to the user. WinHIPE's visualization mechanism focuses on expression evaluation, which matches the nature of the functional paradigm - programs are generally structured as value computations rather than sequential statements. The expression evaluation model in WinHIPE is based on *term rewriting*, which involves the successive replacement of terms in the expression working toward a normal form (i.e. the value of the expression). The tracing system supports a number of ways to step through the evaluation of an expression, including one step at a time, n steps at a time, or all at once. Moving backward through evaluation is also supported. WinHIPE also introduces a form of breakpoints so that a function may be marked and evaluation is paused at the beginning of each evaluation of that function.

The expression evaluation model in WinHIPE is based on *term rewriting*, which involves the successive replacement of terms in the expression working toward a normal form (i.e. the value of the expression). The tracing system supports a number of ways to step through the evaluation of an expression, including one step at a time, n steps at a time, or all at once. Moving backward through evaluation is also supported. WinHIPE also introduces a form of breakpoints so that a function may be marked and evaluation is paused at the beginning of each evaluation of that function.

Animations in WinHIPE are shown as a sequence of static frames, each corresponding to a specific point in the evaluation of an expression. Because some evaluations may involve a very large number of individual steps, the user is allowed to manually select which frames should be included in a customized animation and can save the animation for later. Once the frames to include are selected, each is labeled with the individual step of evaluation that corresponds to the transition from the previous selected frame. Textual explanations for each frame may be manually annotated and included in the animation. The final animation can be stepped through manually (both forward and backward), or played at a specified speed.

By default, animations in WinHIPE are displayed so that every part of intermediate expressions are visible to the user, but very large expressions can be shown using a customizable "fish-eye" view [82] that automatically elides subexpressions that are less relevant to the current evaluation. Most expressions are displayed using a textual format similar to the syntax with which they are originally written, but specialized graphical representations are used for lists and trees because

textual representations proved unreadable. Fully evaluated subexpressions are distinguished from those that have yet to be evaluated. Aesthetic components of the visualizations (e.g. font, spacing, colors) can be customized by the user.

CHAPTER 4

Illuminating the Notional Machine

Illuminating the notional machine involves the use of verbal and visual media to provide students with information about what the notional machine does. In a traditional setting, the notional machine is essentially a black box to students. The limited feedback they do get from program output (or error messages if something goes wrong) does little to help them learn about what their program is actually doing at runtime, because students face a prohibitive attribution problem. We seek to eliminate this problem by using program visualization and a means to expose the program as a dynamic entity as it runs on the notional machine, so that students may experience running a program as an interactive, hands-on experience with meaningful feedback. Figure 4.1 diagrams the big picture of our approach.

The importance of developing a mental model of the notional machine – in order to really understand what is going on inside a program – informs the approach of this thesis. In addition to developing an accurate understanding of language constructs and how they work together in the context of the notional machine as a whole, students must also develop problem solving skills in this context - problem solving "on the notional machine". Approaches without this focus are less effective: problem solving in the space of syntax and written code is a fragile approach that fails to generalize, and problem solving in the problem space is often not troublesome in the first place, at not least for the simple tasks encountered when first learning to program.

4.1 Notional Machines and Black Boxes

In programming, a "notional machine" is an idealized, hypothetical machine on which programs run [12]. A notional machine serves as an abstraction to explain the way programs execute, and a viable mental model of the notional machine is a prerequisite for effective programming. The workings of a notional machine are implied by the constructs of a particular programming language and operate on a conceptual level - they need not be tied to any real compiler or hardware. This means that concepts illustrated on a notional machine can be presented strategically to help students

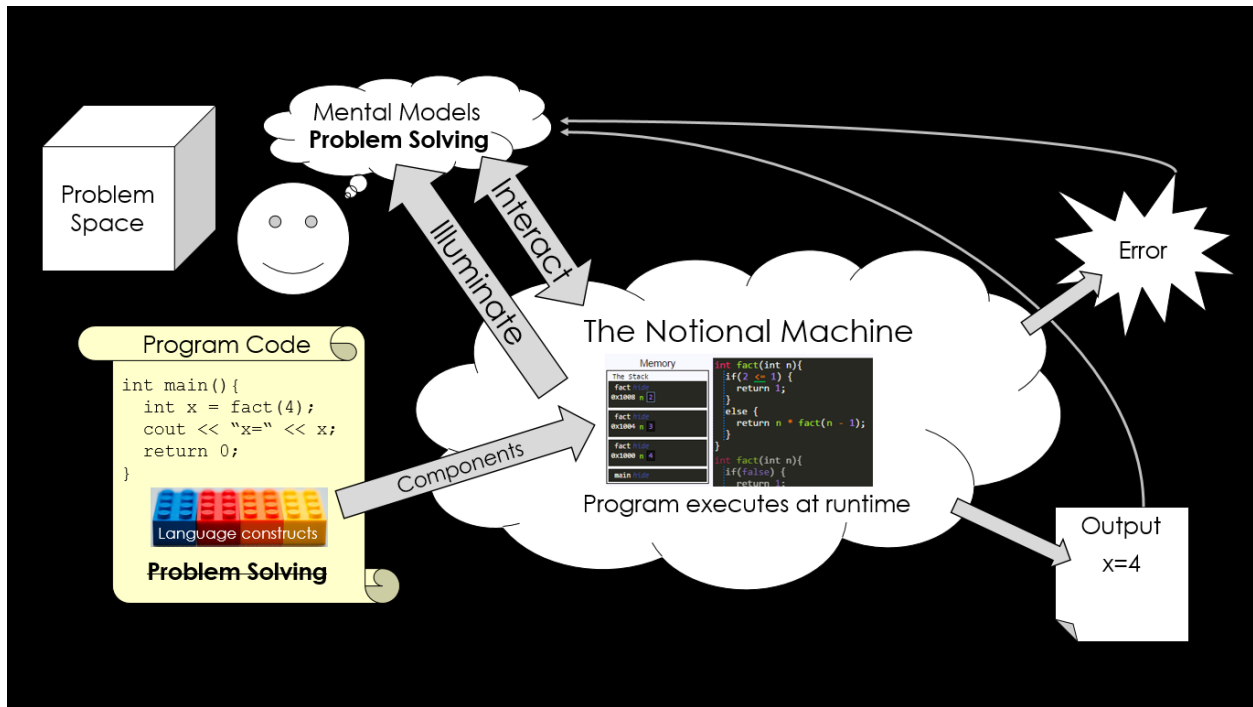


Figure 4.1: A diagrammatic view of the relationships between the problem solving process, the notional machine, and students' understanding. The goal is to encourage and provide the necessary tools for students to perform problem solving with viable mental models of the notional machine rather than as an exercise in writing the correct program code to be run on a black-box. Program output and runtime errors provide very little feedback from which the student can learn, and suffer from attribution problems. We illuminate the notional machine and allow students to interact with it in order to support the active construction of mental models. The way individual language constructs come together to form a dynamic program as a whole is evident. Finally, the feedback students receive is much richer, does not suffer attribution problems, and is responsive to the students' individual mental models as they develop.

understand program execution. For example, students first learning subtype polymorphism can more easily approach the idea of a conceptual machine that uses the most derived type, without the details of a particular implementation using a vtable. Several notional machines at different levels of abstraction may also be used.

Unfortunately, many students only experience the notional machine as a "black-box". A student in a traditional setting writes a program, attempts to compile it, and runs it - the only pieces of feedback they receive are compiler errors/warnings and program output. Neither of these provide much, if any, insight into what a program actually does when it runs. At best, students are left trying to infer dynamic properties of their programs from static, textual code and verbal descriptions of what it should do. Without a window into what is happening at runtime, students' construction of mental models of the notional machine is hindered.

When learning to program, students may derive some meaning from the metaphors implicit in the names of language constructs [12], but unfortunately these metaphors often do not have enough fidelity to support the creation of viable mental models. This can lead to fragile understandings dependent on when the metaphors happen to match up with reality. Another pitfall in learning from a black-box model of a system is that superficial features may be granted inappropriate or misplaced significance [12]. On the other hand, if students are able to see precisely how constructs and related metaphors translate to a dynamic program executing at runtime, they can avoid these misconceptions.

It is worth noting that the notional machine we are trying to illuminate in program visualization is a much more complex artifact than that which we are trying to illuminate in algorithm visualization. The program being run in the case of algorithm visualization is essentially fixed to focus on a single algorithm of interest. Of course, the input cases to the algorithm may change, but the essential procedure that is being carried out is the same each time, and the visualization can be tailored to illustrate the way this specific procedure works. This is not the case with program visualization, because what we want to visualize is a conceptual machine that is capable of running any program. Many design choices that come up in illuminating the notional machine through program visualization are more complex than those in specialized algorithm visualization systems. For example, the question of which conceptual model to use for subcall execution and how to represent it visually in a program visualization is more complex than the question of how to visualize a swap in a sorting algorithm.

4.2 Thinking About Dynamic Programs

There is a significant phenomenographic difference between thinking about programming just as writing code or as thinking about the dynamic runtime entity defined by the code [83] and which runs on the notional machine. For example, in addition to simply perceiving written code, it may be second nature for experienced programmers to mentally trace through what their code will do when run. Sorva [84] proposes that an understanding of program dynamics - the essence of a program as a “dynamic execution-time entity within a computer” and not just as static code - may well be a *threshold concept* for programming students. If students learn to think about programs as dynamic entities, they are enabled to approach programming from a different and more effective perspective. Put another way, students must first master program dynamics before they can begin to truly “think like a programmer”.

A threshold concept [85] represents a particular learning objective that can fundamentally change the way students approach a topic in general. Threshold concepts are often *troublesome* for students due to a number of reasons. The concepts themselves may be particularly complex or may

not be intuitive to new learners. Another facet of threshold concept theory is the *liminality* of the transition. Crossing the threshold marks a qualitative difference in understanding and is not just a matter of degree. Once mastered, an understanding of the threshold concept enables the student to much more easily approach subsequent learning goals. It is also not easily forgotten because the “learning” involves a fundamental shift in the way one thinks about the material.

Focusing on threshold concepts when designing curricula enables instructors to spend time efficiently both because the concepts themselves are important and because they enable students to think in a way that enables them to pick up other concepts more quickly. Teachers must be intentional about teaching threshold concepts because as experts it they may not always be able to relate intuitively to the thinking of others who have not crossed the threshold.

A general characteristic of threshold concepts is that they often tie together many different aspects of knowledge in a particular field that may seem unrelated on the surface. This is certainly true for program dynamics - all the various coding patterns students learn for solving different problems are related by the mechanics of code execution. An understanding of program dynamics illuminates why subtle changes in code can lead to a profound difference in the way the code actually runs.

4.3 Mental Models

When people try to learn about an artifact, they develop an internal mental model of how it works based on the interactions and experiences they have with the system in question [86]. For our purposes, a crucial goal of programming education is for students to develop a viable mental model of the notional machine. A *viable* model can be used to make accurate predictions about the system’s behavior in a variety of situations, and for the notional machine this amounts to students ability to comprehend what programs do when they are run. In this sense, students can “run” code on their mental models in order to mentally trace code, but this skill requires a well-developed model and is often quite difficult for novices. On the other hand, models that sometimes make incorrect predictions or do not “run” smoothly are called *fragile*.

Mental models cannot be taught in a direct sense. Rather, they are constructed by learners in response to their experiences with the target material. The construction process is gradual, and models evolve over time to account for each new experience a learner has with the target system. A mental model often does not evolve to literally encode the way the system actually works internally, and the mental models formed by individual learners will often be different from each other.

A conceptual model may be used as an education resource to present a particular representation of the system at hand in order to provide students with experiences from which they can construct their own internal, mental models. The conceptual model is not intended to be adopted directly by

the user as their own mental model, but rather to provide the ideal view of the system at hand for mental model construction to take place. Conceptual models often leverage abstractions to hide away unnecessary details and help learners manage cognitive load. Norman discusses Three goals for effective conceptual models given by Norman [86] are:

- **Learnability** The student must be able to understand the model that is presented to them and readily develop a mental model in response.
- **Functionality** The conceptual model must accurately represent the behaviors of the target system so that the mental models students develop will in turn be able to make accurate predictions about the system's behavior.
- **Usability** The conceptual model must provide a mechanism for reasoning about the target system within the constraints of human cognitive processing.

4.3.1 Mental and Conceptual Models of the Notional Machine

It is generally well accepted that a mental model of the notional machine is essential to effective programming (see e.g. [87, 88, 89, 13, 14]). However, as Ben-Ari argues [13], students do not innately possess an effective model of a computer. In programming, there is often no clear “real-world” parallel to a particular concept, and the analogies students tend to apply by intuition often lead to non-viable mental models. For example, many students will not have prior experience thinking with an explicit level of indirection, but in languages that use pointers rely heavily on the concept. Even as they learn, novice programmers' mental models of a notional machine are likely to be “incomplete, unscientific, deficient, lacking in firm boundaries, and liable to change at any time.” [14]. As such, developing a viable model of the notional machine must constantly be kept in sight as one of the objectives for students in programming courses.

At the same time, presenting students with a conceptual model that shows parts the notional machine can influence their mental model development. In particular, a visualization of a conceptual model can give a student tangible experiences from which to construct a viable mental model of the notional machine. It is important to emphasize that the visualization is not necessarily presenting a mental model itself, because students must still construct their own models internally. Rather, a visual and interactive representation of the notional machine allows students the kind of active learning experiences from which they can build and refine their own understanding.

In one experiment [90], students who worked with a program tracing tool tended to develop more semantically-oriented mental models while other students who worked without the tool tended to form syntactically-oriented mental models. This makes intuitive sense, because the students who used the tool were in a way interacting with the notional machine, which defines the

semantic meaning and runtime behavior of language constructs, whereas the other students were only experiencing the language constructs at a syntax level.

In another experiment [91], students were tasked with learning how to write simple programs in a BASIC-like language. The treatment group was given a conceptual model of the computer, whereas the control group was not. Both groups then read an instruction manual describing the language, which contained both explanations of the language constructs and examples of their use. The students were then given a code-writing test. On problems that were very similar to the material in the instruction manual, the control group performed just as well or better. However, on problems that required at least a moderate amount of transfer (i.e. they were unfamiliar tasks for the students but could be solved using the same knowledge), students who had learned with a conceptual model performed better. A later study [92] found that presenting the conceptual model to students after they have already finished studying the instruction manual did not significantly impact learning outcomes.

These results are consistent with the underlying idea that while a conceptual model of the notional machine may not improve straightforward retention of material or improve learning on its own, it can provide a context for knowledge construction that leads to a more robust, viable mental model that can be more readily applied to new situations.

4.4 Working with Own Code

A major component of our approach is that students are able to perform any visualization and interaction activities with code that they themselves have written. Students will be fundamentally more invested in content they have had a hand in generating, and the experience they have with the visualization is responsive to their current mental model which was used to write the code. Misconceptions that students hold will be exposed when their code is run and are able to refine their understanding in response. Allowing students to freely work with their own content also enables them to explore what-if questions as they come up and removes the sole reliance on an instructor preparing a small set of examples ahead of time that must somehow be relevant to all students.

A visualization system that allows students to modify the code being visualized enables them to explore what-if questions as they come up. Examples prepared ahead of time by an instructor, even if carefully selected, cannot reasonably account for all cases a student might be interested in. Beyond this, as the number of examples provided increases, students may become overwhelmed or have difficulty determining which to focus on. Of course, instructors may still provide an up-front set of examples known to be tricky or that highlight important cases, but students now have the freedom to explore beyond these in response to their own understanding. What-if type questions

often arise in response to a student's own realization of a gap in their knowledge, and allow the opportunity to fill these kinds of gaps.

Allowing students to explore the space of possible programs on their own also gives opportunities for students to pursue higher level learning objectives in Bloom's taxonomy. For example, support for own content provides an environment in which *creating* can take place. Students can write their own programs, use the visualization to inspect and reflect on what they have created, and iterate back to make changes based on what they saw. Evaluating is also in view. For example, a student might be asked to experiment with different ways to code a particular example and weigh the pros and cons of each. The visualization can be used as a tool to look inside the black-box and gain an intuition for what the code really does, which feeds into forming deeper and more meaningful evaluations of the code in question.

The content dimension of the 2DET includes *given content*, *own cases*, *modified content*, *own content* (in order of increasing degree of ownership). A program visualization system that allows the highest levels of content ownership is able to offer much more for students learning. Essentially, students learn more by doing than by watching. Students working with their own content will be more invested, have the opportunity to explore what-if questions, and are able to work on higher-level learning objectives.

The rationale given by Sorva et al. for including a separate dimension for content ownership is in part that learners may be more invested in and motivated by content they have had some hand in creating, but their formulation also provides clarity lacking in some parts of earlier taxonomies. Any learning activity with a visualization naturally includes some degree of content ownership and a one dimensional taxonomy cannot accommodate this well. For example, the original engagement taxonomy [16] did not thoroughly account for content ownership and thus whether students view a visualization of given code, code they are allowed to change, or entirely their own code (potentially very different learning activities!), the level of engagement would just be *viewing*.

The extended engagement taxonomy of Myller et al. attempts to address content ownership by adding the *entering input* and *modifying* categories, but these are awkward to consider as a full learning activity on their own - they must be paired with some other mode of engagement to make sense. For example, students might enter input and then engage in controlled viewing of the result, but it doesn't make sense to just enter input! This phenomenon speaks to the need for an additional, orthogonal dimension so that one always considers a particular pairing of both direct engagement and content ownership to characterize a learning activity.

4.5 Techniques for Illuminating the Notional Machine

The goal of illuminating the notional machine so that students can have rich, hands-on learning experiences, especially with their own code, provides a lens through which we can view many techniques for educational program visualization. This perspective puts previous works in context, many of which have investigated isolated aspects of program visualization, but it also exposes gaps in the literature. In this work we fill some of these gaps with novel techniques for program visualization, including in-situ expression evaluation, point and click navigation, targeted error messages, code-context visualization, static analysis/feedback and runtime analysis/feedback. In our overall approach, we draw on all of these different aspects of program visualization working in concert to help students learn about the notional machine. Figure 4.2 shows a brief overview of each of these aspects.

Aspect	Examples from Previous Works	Description
Dynamic-Logical Model for Subcalls [80]	EROSI [93]	Subcalls are illustrated as a separate, unique instance of the called function with it's own code/memory display.
Forward/Backward Navigation	Online Python Tutor [79], VILLE [94], WinHIPE [72]	Step-by-step navigation is possible forward and backward from any point in the simulation.
Memory/Code Duality	The Teaching Machine [95]	Visualization of the execution of code and a display of memory offer complementary views of the notional machine.
Multiple Representations of Data	HDPV [96]	Representations of values and objects are designed to highlight the most important details of how the notional machine processes them.
Completeness and Continuity [81]	Jeliot, others [78]	The visualization shows every relevant step taken by the notional machine and should make clear the transitions from one state to another.
Comprehensiveness	Many Systems	Support a comprehensive "language" to offer an authentic programming experience and avoid disrupting students' exploration of the notional machine.
No Barriers to Use	Guo [79]	Eliminate barriers to student use (e.g. up-front installation, training requirements).
Code-Context Visualization	Novel	Use a dynamic representation of the code itself as an authentic visual/interactive medium (rather than visual metaphors) to illustrate a program's execution.
Notional Machine-Targeted Error Messages	Novel	Both compile-time and run-time error messages describe what the notional machine was trying to do and what the problem is.
In-situ Expression Evaluation	Novel	Expression evaluation is shown step-by-step as an incremental process of replacing each subexpression with its computed value and in-place, rather than in a separate area.
Point+Click Navigation	Novel	Navigation to a particular point in the simulation by clicking on the code itself, rather than only via forward/back buttons.
Static Analysis/Feedback	Novel	Static analysis identifies conceptual patterns in students code and explains them to students in terms of their own code.
Runtime Analysis/Feedback	Novel	Dynamic analysis identifies errors or undefined behavior while a program runs and alerts students with an explanation and guidance for investigating the problem.

Figure 4.2: Aspects of educational program visualization that support the principles of learning about the notional machine. Some appear previously throughout the literature, and the middle column gives example systems which exemplify the approach (these are not intended to be exhaustive). Others are novel techniques presented in this work. Each of these techniques is supported in the Labster system, and we describe experimental evaluations of several in this dissertation.

CHAPTER 5

The Labster System

Labster is a web-based program visualization system that supports working with students' own programs in the C++ language. No installation is required, and using it is as simple as visiting a web-page. The system supports nearly the entire C++ language and generates visualizations effortlessly. We have integrated the use of Labster into the EECS 280 course at the University of Michigan as part of select lab exercises and in-class activities, but also for students to use on their own to explore programming concepts. As well as an educational tool itself, Labster also serves as a research platform for investigating program visualization techniques, since the system architecture can easily be adapted or extended as desired. Labster provides support for each of the techniques in figure 4.2 and we have conducted experiments to evaluate the effectiveness of several, including the conceptual models for expression evaluation and subcall execution, as well as the controls for navigating through the visualization. We discuss those experiments in later chapters.

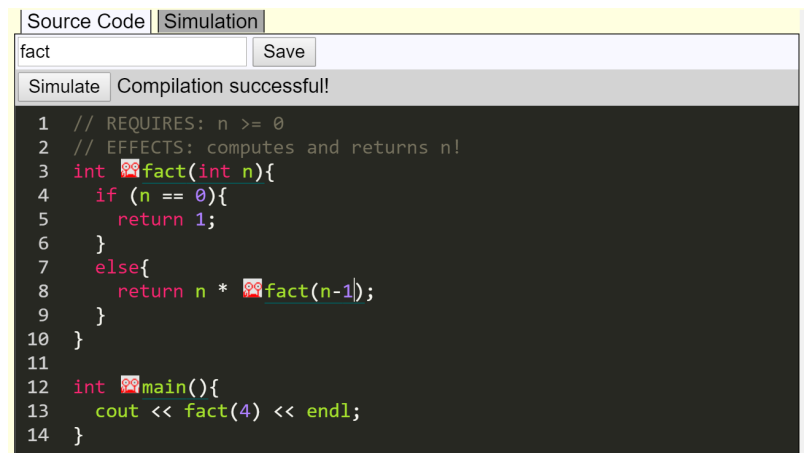
In this chapter, we provide a description of several features of the Labster System and discuss educational considerations that inform their design and the way in which they support student learning.

5.1 Narrative Example of Using Labster and Code-Context Visualization

In this section, we will give a narrative account of the way a student might use Labster in order to provide the reader an introduction to the system and the way the design principle of *code-context visualization* informs the *code stack* model for subcall execution and the “in situ” expression evaluation used in Labster.

The system allows students to write and work with arbitrary C++ code, but also can be configured to provide prepared code that can serve as examples or starting points for students to make their own modifications. In this case, we assume the student is starting with the recursive implementation of the `fact` function shown in Figure 5.1, which computes the factorial of a given

number. The implementation is split into a base case and recursive case where `fact` calls itself as a subroutine to compute the desired result.



```
Source Code | Simulation
fact Save
Simulate Compilation successful!
1 // REQUIRES: n >= 0
2 // EFFECTS: computes and returns n!
3 int fact(int n){
4     if (n == 0){
5         return 1;
6     }
7     else{
8         return n * fact(n-1);
9     }
10 }
11
12 int main(){
13     cout << fact(4) << endl;
14 }
```

Figure 5.1: A recursive implementation of factorial, provided as starter code in Labster. Students can work with a visualization of the code as given, but can also make any changes they like to their own copy of the code and explore resulting effects on the program’s behavior.

In order to view a visualization of the current code, students switch to the visualization view which initially only shows the `main` function in the code execution area and a corresponding stack frame for `main` in the memory diagram. Other functions do not appear until they have been called. The piece of code which is “up next” (i.e. just about to be executed) is highlighted either with colored underlines or backgrounds, depending on the type of the construct (e.g. green underline for expression evaluation, colored backgrounds for implicit conversions). In our example, the very first step is that the `fact` function is just about to be called. As we progress through the simulation, each call to `fact` produces a new invocation of the function in the code area as well as a new stack frame in the memory diagram (See Figure 5.2.). When each invocation returns, it is removed from the visualization. The parallel stacks of code execution as well as memory are the major pieces of what we call the *code stack* model for subcall execution.

As the visualization progresses, several expressions must be evaluated. As each one is stepped through, its evaluation is animated “in situ” – in the natural setting of the code itself. Essentially, the code is “brought to life” as if it were a dynamic entity in the notional machine at runtime. For example, when testing whether the base case should be applied, the expression `n == 0` is incrementally visualized first by looking up the current value of `n` and subsequently turning into either `true` or `false`. Another expression, `n * fact(n)` has its evaluation “interrupted” by a function call. The value of `n` is first looked up and remains displayed throughout the execution of the subcall, which provides valuable context for the overall execution of the program. Only after the call to `fact(n)` has completed and has been replaced by the returned value does the

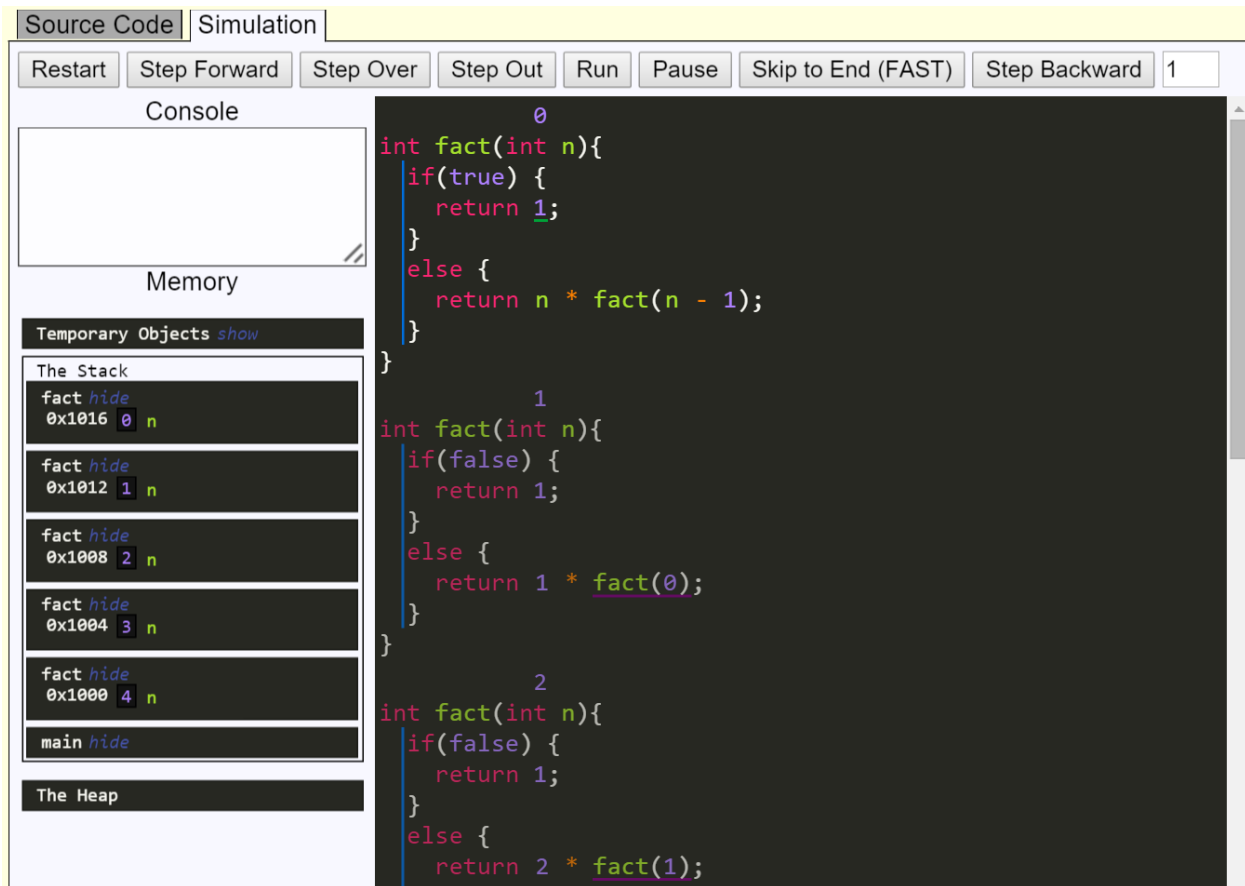


Figure 5.2: A snapshot of the visualization for the `fact` function. The code execution and memory displays in Labster parallel each other in accordance with the *code stack* model of subcall execution and several expressions which have been partially evaluated “in situ” are visible.

multiplication take place.

The interplay between “in situ” expression evaluation and the *code stack* model allows for context in the visualization of where the program will be going in passive control flow. In this particular implementation of the `fact` function, all of the computation is interleaved in the passive control flow. That is, as the program actively creates new invocations of the factorial function, the current value of the parameter `n` to each is essentially stored on the stack frames that have accumulated and none of the multiplication happens until the call stack unwinds. The fact that all the invocation of the recursive function are shown stacked up in the code execution area (i.e. the *code stack* model) and that each shows partially evaluated expressions including the pending multiplications (i.e. the expressions are “in situ”) means students can clearly see and gain an appreciation of how the recursive code actually performs the overall computation.

5.2 Situating Labster

With respect to the taxonomy of effortless visualization of Ihantola et al. [22], Labster’s scope is somewhere between course-specific and domain-specific. At present, Labster only supports the C++ language, but this is sufficient to support use in several different programming courses as well as data structures and algorithms courses. The ubiquity of programming throughout computer science means that program visualization is inherently applicable in many situations. Considering the categories of the *integrability* dimension of the taxonomy, Labster supports easy installation, customization, platform independence, documentation, and integration of hypertext. Support for internationalization, interactive prediction, and course management are being considered for addition in the future. We discuss support for interaction below.

5.2.1 Ease of Use

No program visualization has seen widespread use [18], and surveys of computer science faculty [16] reveal that although they overwhelmingly believe visualization can be helpful for learning, the up-front cost to integrate and use visualization systems in their teaching is prohibitive. Labster is designed to be used anywhere and by anyone, and requires no installation. Using it is as simple as opening a webpage. All of Labster’s coding and simulation features run on the client-side, which means it can be used even without a constant Internet connection. This also means there are no back-end scalability issues to stand in the way of widespread dissemination. Labster is well-suited for use in both traditional or online courses, and can be used for demonstrations during instruction, collaborative work, interactive examples and assignments, or individual coding practice.

Labster supports a subset of C++ that covers almost all features relevant to an introductory programming course. This is important – our experiences with students in the early phases of Labster’s development revealed that students expect a system will support the whole language. If they are trying to write code in a particular way, but one of the constructs they use is unsupported, it is a significant distraction to have to implement a workaround (e.g. using $x = x + y$ instead of $x += y$). Our goal is for Labster to support individualized learning experiences which are responsive to each student’s mental model, and that means it needs to work in all cases.

5.3 Editor and Compilation

When students log into Labster, they have access to several code examples from the EECS 280 lectures as well as starter files for the lab exercises that use Labster. Once students load a file for the first time, a personal copy is created that will save any changes students make to the code.

```
Source Code | Simulation
program.cpp Save
Semantic error(s) detected.
1 int main(){
2     int a = 40;
3     int b = "hello";
4     int c;
5     cout << c << endl;
6     }
7
8
9
```

Line 4: The local variable c is uninitialized, so it will start with whatever value happens to be in memory (i.e. memory junk). If you try to use this variable before initializing it, who knows what will happen!

Figure 5.3: An error is shown on line 3 because the type of the initializer expression is incompatible. A warning is shown on line 4 because `c` is uninitialized. When the user hovers the mouse over the error icon on that line, a detailed explanation appears.

Individual programs can also be started from scratch and saved in the students personal files.

Students are provided a text area for writing their source code. We use the CodeMirror editor [97] as a front end to provide students with syntax highlighting, line numbers, and other niceties expected in modern code editors. As students write code, Labster attempts to recompile on the fly and shows compiler errors and warnings by underlining the relevant part of the code. If the user hovers the mouse over the error icon, a textual error message is shown.

Labster’s compiler messages often give additional insight into the reason “why” the given code is erroneous. For warnings in particular, an explanation is given for the potential problems that could result from the problematic code (see figure 5.3). In some cases, Labster performs additional analysis on the context in which an error appears to provide a more tailored error message. For example, figure 5.4 shows two cases where the context in which incompatible types are found is relevant to the conceptual error a student likely made. In the first case, the student is directed to think about pointers vs. objects rather than just the fact that the operator is not applicable for those types. In the second case, the student makes a classic error by forgetting to account for the null character automatically added to a string literal in certain contexts. Labster detects that the character array is precisely one element too short and brings this to the user’s attention.

5.4 Visualization

Once students have written a program that successfully compiles, they can launch a visualization of their program and observe its runtime execution. Students are provided with a visualization of both the code as it executes and a diagram of the contents of memory.

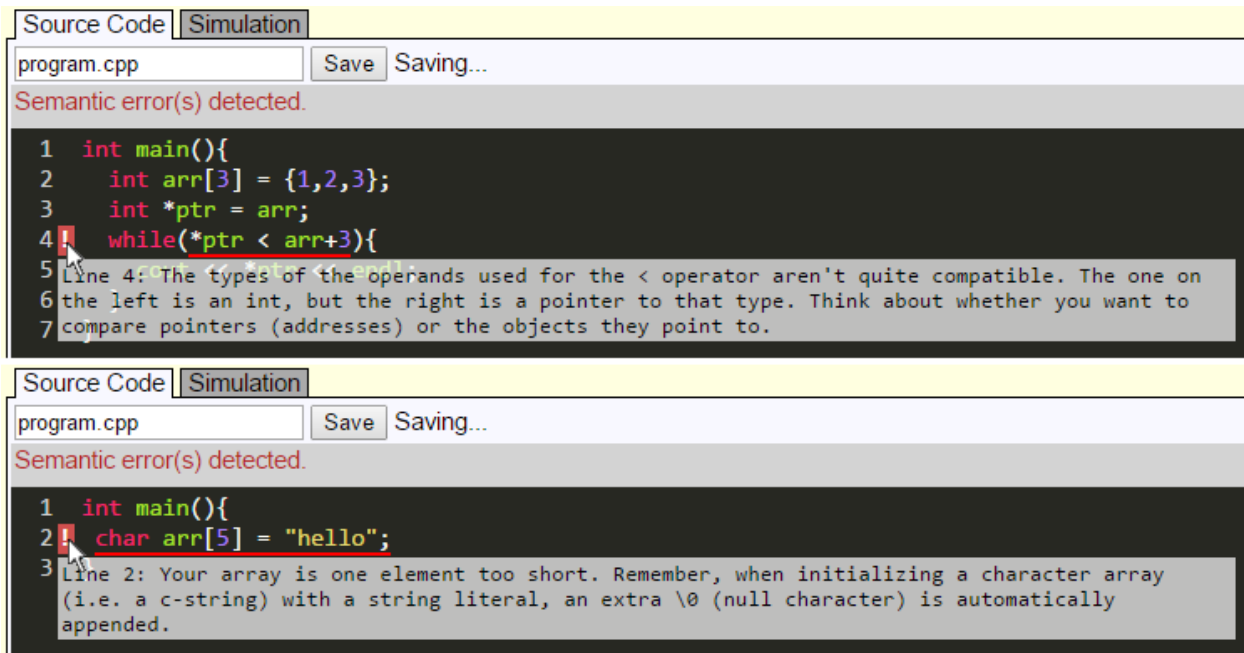


Figure 5.4: Labster identifies particular cases in which additional guidance can be provided to shed light on the conceptual issues behind a particular error.

5.4.1 Interaction Modalities

Labster supports several different interaction modalities. First, we consider activities focused primarily on code reading and program comprehension. This also includes knowledge of language constructs and their semantic meaning. Students might be presented with ready-made examples that illustrate particular language constructs or techniques. They would be expected to step through the program at a varying pace, more quickly for parts they understand well and more slowly for new or complicated code.

Occasionally, a student will observe behavior that surprises them or contradicts what they would expect to have happened. In these cases, stepping backward or rewinding to an earlier statement or the beginning of a function will prove useful for a student to conveniently replay through tricky sections of code. On subsequent runs through the same section of code, students may pay more attention to different parts of the visualization like the memory diagram or auxiliary textual descriptions of what operations are taking place.

Another option would be to support mental program tracing exercises in which students are asked to predict the output of a program. If the student responds incorrectly, then they can view the correct answer but more importantly also have the option to simulate the program and find precisely the reason their prediction was incorrect. This allows for much more fine grained feedback than if students were simply shown the correct output after making their prediction.

In addition to activities focused on developing an understanding of how programs execute, students may also spend time in an iterative process of writing code and then testing it out in the interactive simulation. As soon as a program compiles, students are able to run their program either all at once (to check output) or interactively. They may either start from the beginning, jump to a particular location, or unit test individual functions with adjustable parameters. The latter feature should be particularly helpful for incremental testing because it allows students to focus the simulation on working with the particular piece of code they are currently writing. Being able to easily test a function with customizable inputs allows students to quickly form an intuition of the dynamic relationship between each parameter and the resulting behavior.

5.4.2 Navigation and Controls

Labster offers students the ability to easily navigate throughout the execution of their program.

- **Restart** Restarts the entire program at the beginning of `main`.
- **Step Forward** Moves one step forward in the simulation. This might mean performing the next value computation in an expression, storing a value into memory, or beginning a new function invocation, depending on context. Pressing the right arrow key also activates this command.
- **Fast Forward** When the user hovers their mouse over an upcoming line in a block of code, an arrow appears to the left of the line. Clicking this arrow immediately runs the simulation up to that statement.
- **Rewind** When the user hovers their mouse over a previously executed line in a block of code, an arrow appears to the left of the line. Clicking this arrow immediately returns the simulation to the point of execution at the beginning of that statement.
- **Step Backward** Moves one step backward in the simulation (i.e. inverse of step forward). Pressing the left arrow key also activates this command.
- **Step Over** Runs the next code structure (e.g. full-expression, loop, function call, etc.) to be executed in one step.
- **Step Out** Runs the rest of the currently executing code structure (e.g. full-expression, loop, function call, etc.) in one step.
- **Run/Pause** When the user clicks the run button, Labster enters auto-run mode and will step forward repeatedly until the user clicks the pause button, a special pause function (see below) is executed, or a runtime error occurs.

- **Skip To End** Skips completely to the end of the simulation without displaying animations and ignoring any pauses or non-fatal errors.

The user may also scroll using the mouse wheel to move through the code's execution. Each scroll unit is the equivalent of a step forward command. This provides a much smoother feeling than repeatedly clicking the step forward button.

5.4.3 Navigation Labster Functions

Labster provides a few special functions that can be called in the program code and affect aspects of the simulation when executed.

- void **assert**(bool condition) Causes a run-time error message to be displayed if condition evaluates to false. This also pauses the simulation.
- void **pause**() The simulation will pause if this function is encountered while it is in auto-run mode.
- void **pauseIf**(bool condition) The simulation will pause if this function is encountered while it is in auto-run mode and condition evaluates to true.

5.5 Runtime Feedback

Some situations that may be encountered while running a program merit additional feedback beyond the regular visualization. For example, cases where the behavior of the program is undefined (e.g. inappropriate memory accesses) are immediately brought to the user's attention and briefly explained. As the problematic code is executed, a non-modal pop-up message is displayed.

Runtime alert messages are written in a first-person style and shown alongside a picture of the "Labster Lobster" in order to personify the feedback given to students. This is intended to make learning gleaned from catching the mistakes more memorable. It may also have a positive impact on students motivation to fix these bugs as part of programming exercises [98].

Labster is able to identify several different kinds of undefined behavior at runtime and gives students explicit feedback about what went wrong. This is in stark contrast to the traditional pattern of a student writing code, running it on a computer (i.e. on a black box), and perhaps receiving no specific error messages at all or ones that don't relate directly to the source of the problem. For instance, figure 5.5 shows a simulation of code that accidentally steps over the end of an array and modifies memory it shouldn't. Any feedback to the student is often delayed until some other part of code later tries to use the corrupted memory and results in a mysterious runtime error or

worse, just produces inexplicably incorrect results. In a Labster simulation on the other hand, the student will receive an error message the moment an operation like pointer arithmetic or indexing goes off the end of an array. After being alerted to the problem, students may observe the pointer's position in the memory diagram or choose to step backward to find observe the steps that led up to the problem.

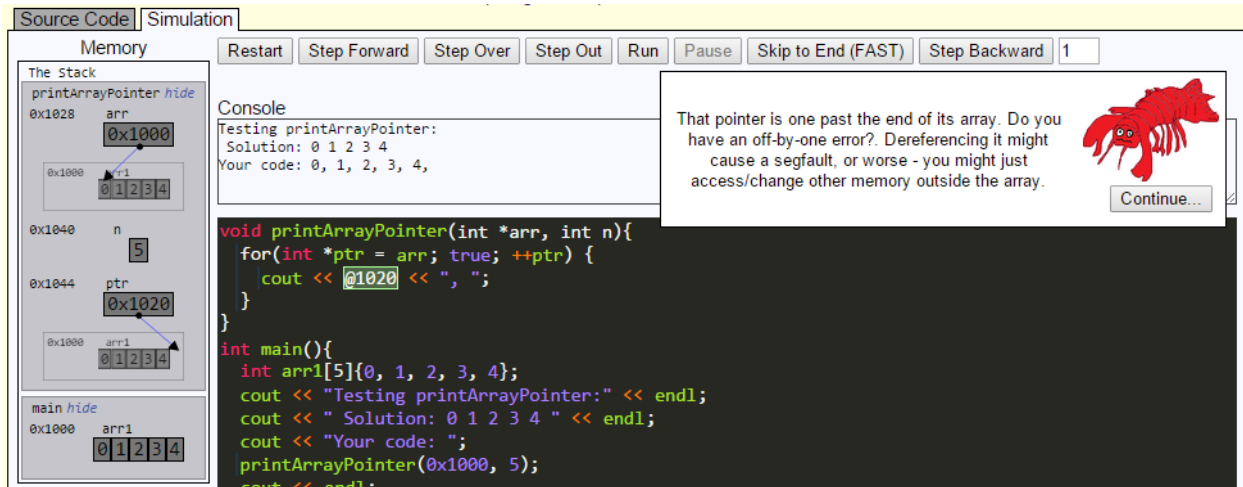


Figure 5.5: The Labster lobster reports operations that have undefined behavior to the user. In this case, the loop condition (which is shown evaluated to true) incorrectly used `ptr <= arr+n` and led to an attempt to read memory outside the array in the body of the loop (which uses `*ptr`).

5.6 Novelty

Many program visualization systems have been developed throughout the last few decades, but our approach and the Labster system are unique in a number of ways. To our knowledge, it is to date the most feature-complete program visualization system that allows student to work with their own content in C++ programming.

The choice of C++ is significant for a number of reasons. C++ programming involves concepts absent from many other languages like value semantics, indirection (pointers), low-level arrays, and explicit management of dynamically allocated memory to name but a few. In particular, topics related to pointers and memory are rated among the most difficult by students and teachers. [99] Additionally, arrays, pointers and references, and parameters are rated as more difficult in C++ than in Java. [100] Certain aspects of C++ (e.g. a more nuanced memory model, separation of value/reference semantics) which make these and other concepts inherently more complex. As such, there are unique opportunities to explore the educational impact of interactive program visualization.

Labster supports nearly all C++ language features that would be relevant for students in programming courses. Many previous systems did not support all of the most common language features (e.g. no support for classes) or had visualizations that lacked completeness or continuity for particular features (e.g. no memory diagram is shown, expression evaluation is not shown step-by-step).

The majority of existing program visualization systems were designed for and evaluated in the context of first programming courses (CS1). While EECS 280 overlaps with some of the material traditionally within the purview of CS1, it also includes several more advanced topics that may not be covered fully until a second course (CS2). From an official course description,

“EECS 280 “Programming and Introductory Data Structures” is a second-semester programming course. It focuses on computer science concepts that are widely applicable to many programming languages, and implements them in C++.

Techniques and algorithm development and effective programming, top-down analysis, structured programming, testing, and program correctness. Program language syntax and static and runtime semantics. Scope, procedure instantiation, recursion, abstract data types, and parameter passing methods. Structured data types, pointers, linked data structures, stacks, queues, arrays, records, and trees.”

We thus have an opportunity to explore the impact of program visualization in teaching and learning about these more complex concepts and programming techniques, many of which are underrepresented in the present literature.

CHAPTER 6

Experiments and Methodology

Throughout this work, we present the results of several experimental educational interventions as part of an ongoing, large-scale study of the use of Labster in EECS 280, a second-level programming and introductory data structures course at the University of Michigan. Students used the tool formally during select lab exercises and for interactive exercises during one of the lecture sections, but it was also available for personal use at any time. Logging data is automatically collected while students use the tool for analysis of how and how much they use it.

6.1 Participants and Data Collection

Students taking the EECS 280 course at the University of Michigan are asked to participate in the study and given a clear option to opt-out. Students who opt-out still use the visualization tool and complete any tests or surveys as part of the regular course, but data from these will not be used for the study. To date, students in the Fall 2014, Winter 2015, Spring 2015, and Fall 2015 terms have participated in the study.

Both of the researchers involved in this study have currently or in the past been part of the instructional staff for the course. Once collected, student data is anonymized and only accessible to research staff. Research data was not be available for any purposes related to normal course procedures (e.g. determining grades). Students were informed about the nature of the study and that whether or not they choose to participate in the study will not affect their course grade.

6.1.1 Grouping by Pre-Lab Confidence Levels

We identified groups of participants based on their confidence with the material before going into the lab exercises. Students responses to several Likert-scale questions concerning their confidence with the material (e.g. "I understand the concept of [topic]", "I feel confident I will be able to answer [topic] questions on course examinations") were converted to a numeric equivalent, averaged, and rounded to the nearest whole number to determine a pre-lab confidence level for each

student. A higher number indicates higher confidence. The percentage of students allocated to each confidence group from both terms is shown in Figure 6.1.

We will focus on groups of students with pre-lab confidence levels of 3 and 4 in some of the experimental results that follow. Roughly, group 4 can be considered moderately confident students, while students in group 3 are less confident than average. The distributions of raw, unrounded averages assigned to these groups were similar across terms, so comparing the same group from one term to the next is fair. Groups 1, 2, and 5 represent the extremes, and while it is interesting to consider the implications of program visualization for these students, we do not have a large enough sample of these groups to report meaningful results.

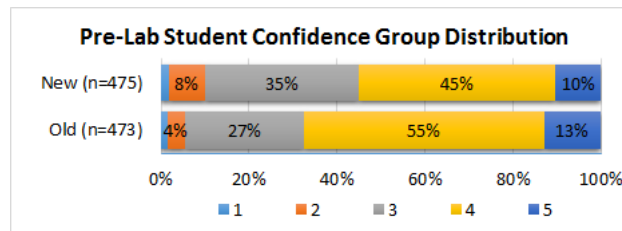


Figure 6.1: Distribution of confidence scores according to pre-test responses. Group 4 represents students who are moderately confident, whereas group 3 students are less confident than average.

6.1.2 Grouping by Time Spent

Basic logging data is collected for user activity on Labster. The time a user spent working with Labster during was measured as the total duration of a sequence of active sessions using the system. An active session starts when a user logs in to the system and ends with a period of inactivity lasting 10 minutes or longer. Any action the user takes (loading a file, modifying source code, stepping in the simulation, etc.) prolongs the current session.

During weeks when Labster was used for lab exercises, we assume the vast majority of time students spent using the system was in fact related to the lab activities. In the Fall 2014, Winter 2015, and Spring 2015 terms, Labster was not used in any official capacity other than the labs. During the Fall 2015 term, Labster was used in one lecture section for in-class exercises, but only after the experiment that term had already been completed.

6.1.3 EECS 280 Terms

Here we briefly describe the participant groups from each term of EECS 280 during which experiments were conducted. We also explain the rationale for excluding the Spring 2015 term from the analyses and results presented here.

6.1.3.1 The Spring 2015 Term

We conducted several experiments during the Spring 2015 term, but ultimately decided not to use these data for our analyses.

We found the Spring term to be too simply too different an environment to make meaningful comparisons with regular terms. The overall pacing of the Spring term is accelerated to fit into roughly 2 months rather than 4 months for a regular term. In particular, students were only given two to three days to complete each lab assignment instead of a week. Additionally, the course staff for the Spring term consisted of only a few veteran instructors, whose individual skill may not be representative of a regular term.

We found that the amount of time spent working on the lab, the amount of time spent taking the surveys, and the percent of students who filled out the surveys incorrectly (e.g. in the wrong order) were vastly different for the Spring term contrasted against the regular terms, which were all similar in these measures. Confidence levels and performance on pre-lab surveys also tended to vary much more when comparing any regular term to the Spring, which may also reflect the different pacing of the course (e.g. which lectures had been covered before/during/after each lab.)

Finally, enrollment for the Spring term is much smaller than regular terms, and only 59 students participated in the study, making it difficult to identify results with statistical significance.

6.2 Methodology

Many of our interventions were conducted under a quasi-experimental, nonequivalent groups design as a part of several lab exercises during the Fall 2014, Winter 2015, Spring 2015, and Fall 2015 terms. These include the results of the expression evaluation experiment in chapter 7 and the navigation controls experiment in chapter 9. In these quasi-experiments, the groups compared were from different terms and received differing conditions with respect to whether Labster was used or which version of Labster was used and which visualization features were available (e.g. what kind of navigation controls were provided, which conceptual model of recursion was visualized).

We also conducted an experimental between-subjects experiment with an independent groups design during the Fall 2015 term. The experiment compared two versions of Labster that presented different conceptual models of recursion, and is discussed in chapter 8. Students in the course were randomly assigned to two groups and had access to one of two different versions of Labster that differed in the conceptual model of subcall execution that was presented.

6.2.1 Lab Exercises

Students complete lab exercises throughout the course, each assigned at the start of the week and due Friday night. Each lab is focused on reinforcing a specific concept (e.g. recursion, arrays and pointers, dynamic memory, etc.) that has been recently covered in lecture. Generally, each lab consists of starter code for an example program and students fill in missing code or function implementations. A set of instructions tells students what needs to be done and guides them toward the correct use of relevant concepts.

Students are encouraged to attend one of several one-hour discussion sections throughout the week during which they work on the lab individually or in small groups. An instructor (TA) is also available during the lab to explain the assignment, review concepts, or answer any questions. However, we have found many students do not consistently attend discussions as the term progresses and prefer to complete the lab exercise on their own time. The labs are designed so that students with a reasonable understanding of the material can complete them in an hour, but many students end up spending additional time.

6.2.2 Pre and Post Lab Surveys

Students who chose to participate in the study took pre-lab and post-lab surveys via an online service. The surveys included short tests to assess their understanding of the relevant course material as well as survey questions about the labs and the use of Labster. The time requirement to respond to the surveys was approximately 10-15 minutes each, although students were allowed to work as long as they liked to complete the problems.

Pre-lab surveys were released to students at the beginning of the week and post-lab surveys were released two days later. In accordance with the way students were allowed to work on the lab exercise on their own throughout the week, many students took the post-survey several days after the pre-survey. Weekly lab assignments were due at 11:55pm Friday, but students were allowed to respond to the post-survey until 11:55pm Sunday. For participants who had taken both surveys, responses were filtered to remove those who had taken the surveys in the wrong order or had taken the pre-survey and post-survey without enough time in-between to actually have done the lab exercises.

6.2.3 Student Confidence

Students responded to Likert scale questions on the pre and post lab surveys about their proficiency with the material for that lab. They were given a set of statements and asked to indicate for each whether they strongly disagreed, disagreed, neither agreed nor disagreed, agreed, or strongly

agreed. The following set of statements, taken from the Arrays and Pointers lab, is representative:

- I understand the concept of arrays.
- I understand the concept of pointers.
- I understand the difference between traversal by index and traversal by pointer.
- I can write code to traverse an array by index.
- I can write code to traverse an array by pointer.
- I can write code to manipulate arrays using pointers. (e.g. Task 2 functions.)
- I feel confident I will be able to answer array/pointer questions on course examinations.

Students' self-reported assessments of learning are often positively correlated with their teachers' evaluations, albeit not completely free of threats to validity [101]. Students self-evaluations are often biased upward, but this effect should apply equally regardless of the educational intervention used. Students' evaluation of the efficacy of particular learning approaches may also be influenced by other factors than the objective degree to which their understanding has improved. For example, enjoyment of a particular learning activity can be negatively correlated with learning, perhaps because activities which do not expose flaws in students learning may make them feel more comfortable [102]. In our experiments, analysis of objective measures of learning can alleviate these concerns.

Furthermore, factors such as confidence and students' attitudes, have been shown to be predictors of student success in CS courses, even in studies where more traditionally held qualifications such as mathematics proficiency or previous coding experience do not correlate as strongly [103, 104]. Wilson [105] found that a student's comfort level was the strongest predictor of success in a CS1 course. Comfort level was judged by a student's likelihood of asking questions during various parts of the course, perceived anxiety while working on assignments, perceived difficulty of the course and programming in general, and perceived understanding of course material as compared to classmates. This result suggests that increasing student comfort level may be a worthwhile goal in its own right, although it must also be cautioned that students who are already set to succeed in a course for other reasons would tend to feel more confident. Additionally, comfort level may also play an especially important role in the success of underrepresented groups in computer science courses. Scragg and Smith [106] found that female students' self confidence exerts an influence on retention, although the evidence was not overwhelming.

6.2.4 Questions about Lab Efficacy

Students responded to several questions about their experience with the lab exercises on the post-lab surveys. Students were given a set of statements and asked to indicate for each whether they strongly disagreed, disagreed, neither agreed nor disagreed, agreed, or strongly agreed. The following set of statements, taken from the Arrays and Pointers lab, is representative:

- This lab has improved my understanding of arrays and pointers.
- The lab activities were an effective use of time.
- I was able to keep up with the pace of the lab and finish on time.
- I enjoyed working on this lab.
- Working on this lab gave me a better idea of how code using arrays and pointers actually works.
- Working on this lab has improved my programming skills overall.

6.2.5 Questions about Using Labster

If the students had used Labster, they also responded to questions about Labster and its use during the lab exercises. Some questions asked about using whether Labster was helpful with respect to specific conceptual topics covered during the lab. Students were given a set of statements and asked to indicate for each whether they strongly disagreed, disagreed, neither agreed nor disagreed, agreed, or strongly agreed. The following set of statements, taken from the Arrays and Pointers lab, is representative:

- The Labster program visualization was clear and easy to follow.
- Using Labster improved my understanding of arrays.
- Using Labster improved my understanding of pointers.
- Using Labster made code using arrays and pointers easier to understand.
- Using Labster made working the lab exercises more productive.
- Using Labster made working the lab exercises more enjoyable.
- I plan to use Labster for additional study or to review arrays and pointers on my own.

- I would recommend Labster to other students learning about arrays and pointers.
- I would like to use Labster for more labs beyond this one.

6.2.6 Conceptual Questions and Coding Problems

We developed a set of questions designed to evaluate students' degree of understanding of the material covered in each lab and proficiency with relevant coding skills. In particular, the code reading/writing questions concern distinct programs and examples from those encountered in the lab material itself. Thus we are able to quantify the degree to which students' learning is able to generalize to other cases involving the same concepts. Additionally, no individual student received the same question on both the pre-test and post-test. We felt that testing students ability to approach a novel problem was a more genuine measure than if they had seen the exact same problem on the pre-test and the post-test.

Some of the problems given to students on the pre-lab and post-lab surveys required students to write a code snippet or function implementation to perform a particular task. These questions were scored according to a rubric which allowed partial credit for each conceptual piece of a working solution. The scores assigned to submissions form an ordinal scale, such that a higher score represents better performance, but numeric differences between scores are not meaningful.

For the Recursion lab, the problems also specified whether the solution should be written using iteration, recursion, or tail recursion. Responses that used the wrong strategy were considered incorrect even if the code was otherwise functional. We did not deduct points for poor formatting or superfluous syntax errors. The two code-writing problems are shown in Figures 8.6 and 8.7.

6.3 Measurement, Statistical Procedures, and Significance Testing

Results from lab surveys are all measured on an ordinal scale and statistical significance testing was done using the Mann-Whitney U test. This includes student responses to five point likert scale agree/disagree statements regarding confidence with the material, statements about the lab activities, and statements about the Labster system. Scores on the conceptual, code-reading, and code-writing questions also form an ordinal scale.

CHAPTER 7

Experiment: Expression Evaluation

Nearly all the “heavy-lifting” done in a program happens in the context of expression evaluation. In order to illuminate the details of expression evaluation by the notional machine, a visualization system must support stepping through the program at a very fine of granularity. However, it is also important to consider the conceptual model that is used to present expression evaluation to the user. The Labster system uses a novel conceptual model of “in situ” expression evaluation. Essentially, expressions are visualized as dynamic source code - as each subexpression evaluates, an animation smoothly replaces it with the resulting value. This all occurs in the same, natural context where students will read and have to understand expressions when they are writing code.

In this chapter, we survey the program visualization literature relevant to expression evaluation, describe the “in situ” expression evaluation model, and report the results of an initial investigation that shows working with Labster improved students ability to trace code using expressions to work with arrays and pointers.

7.1 Background

Sirkia discusses the importance of [107] expression-level visualization and gives several examples of language constructs that require it. These include assignment, function calls, object construction, and member access. Many others exist as well. He also describes difficulties in expression-level visualization including presenting the right level of granularity and finding a way to distinguish between the many different kinds of expressions within the visualization. Technical difficulties also arise if a visualization is based on traces from existing debugging tools which only provide statement-level information. Labster avoids this issue because it works as a full-fledged interpreter for the language and has access to information about the program state at any level of granularity.

7.1.1 Visualization Systems for Functional Programming

Much of the work on visualizing expression evaluation has been done in the context of the functional programming paradigm. Expression evaluation plays an even larger role in functional programming languages than in an imperative/procedural paradigm, because the program is almost entirely structured as a value computation rather than as a sequence of statements producing effects.

At the same time, elements of a visualization like a memory diagram are less relevant in a functional paradigm because the notional machine in functional programming has no concept of state. A conceptual model of expression animation generally does not need to account for side-effects (i.e. anything other than "evaluation"), because functional programming languages generally have no side-effects. Thus, conceptual models that are successful in functional programming may not always translate to other paradigms.

In this section, we give an brief overview of visualization systems for functional languages in the previous literature.

7.1.1.1 The WinHIPE Visualization System

WinHIPE [72], an IDE for functional programming, supports interactive program tracing and animation. It adheres to the effortless paradigm of visualization in that the system automatically generates visualizations for written code and there is no overhead for the user to work with the visualizations. The goal of this approach is to provide high value visualizations at low cost to the user.

WinHIPE's visualization mechanism focuses on expression evaluation using a model based on the idea of *term rewriting*, which involves the successive replacement of terms in the expression working toward a normal form (i.e. the value of the expression). The tracing system supports a number of ways to step through the evaluation of an expression, including one step at a time, n steps at a time, or all at once. Moving backward through evaluation is also supported. WinHIPE also introduces a form of breakpoints so that a function may be marked and evaluation is paused at the beginning of each evaluation of that function.

Animations in WinHIPE are shown as a sequence of static frames, each corresponding to a specific point in the evaluation of an expression. Because some evaluations may involve a very large number of individual steps, the user is allowed to manually select which frames should be included in a customized animation and can save the animation for later. Once the frames to include are selected, each is labeled with the individual step of evaluation that corresponds to the transition from the previous selected frame. Textual explanations for each frame may be manually annotated and included in the animation. The final animation can be stepped through manually (both forward

and backward), or played at a specified speed.

By default, animations in WinHIPE are displayed so that every part of intermediate expressions are visible to the user, but very large expressions can be shown using a customizable "fish-eye" view [82] that automatically elides subexpressions that are less relevant to the current evaluation. The degree of interest required Most expressions are displayed using a textual format similar to the syntax with which they are originally written, but specialized graphical representations are used for lists and trees because textual representations proved unreadable. Fully evaluated subexpressions are distinguished from those that have yet to be evaluated. Aesthetic components of the visualizations (e.g. font, spacing, colors) can be customized by the user.

An empirical evaluation of WinHIPE [62] compared learners using the system to build visualizations of a tree traversal algorithm (i.e. selecting which frames to include in a final animation) to those only viewing animations that had previously been generated by others. This experiment should also be understood in the context of effortless algorithm animation - the primary work of students in the builder group was to determine which frames to include in the animation. This is only a small set of activities normally included at the constructing level of engagement, and the rest were essentially made "effortless" for students, so the results may not extend to other forms of constructing. It may be that the careful consideration of the algorithm required to determine which frames are relevant can be distilled as one of the most impactful elements of construction activities.

7.1.1.2 Other Visualization Systems for Functional Languages

The ZStep 95 system [108] visualizes the execution of Lisp programs and provides unique mechanisms for navigating through the evaluation of expressions. Clicking on an expression in the program display will automatically run either forward or backward to the point at which that expression is executed. A number of other systems that provide debugging and/or visualization support for functional languages have been developed and used in educational contexts, including the LISP Evaluation Modeler[109], ELM-ART [110], and an interactive visualization system for the Miranda language [111].

7.2 *in situ* Expression Evaluation in Labster

The conceptual model of expression evaluation used in Labster is based on a term-rewriting model, but specifically performs term-rewriting in the context of the source code itself. The code for any expression is essentially a dynamic medium for visualization to take place. When the program runs and an expression is being evaluated, each step forward is visualized as a smooth transition

with a particular subexpression replaced by its computed value. The conceptual model is intended to convey the sense of an incremental replacement of subexpressions with their computed value, moving from the inside out.

A key point of the “in situ” model that differentiates it from the conceptual models used in other systems is that expression evaluation is visualized in an appropriate context. When students write programs, and specifically expressions, they will not be writing in a designated “evaluation area” or using visual metaphors of boxes containing values and graphical representations of operations. Rather, they will be looking at source code, and so we use the code itself as a medium to visually present a term-rewriting model. We bring the source code to life to show how it evaluates, in arguably the same fashion an expert programmer’s mental model allows them to trace through evaluation in their head.

Furthermore, Labster allows the user to step through each individual operation in the evaluation of an expression (see figure 7.1), even including things like implicit conversions and the lvalue-to-rvalue conversion responsible for “looking up” the value of an object in memory”. This is essential to uphold the principle of continuity [81] so that students understand exactly how a particular value is calculated or why certain side effects are produced. Stepping through at a coarser granularity would certainly be sufficient activities like debugging when an expert programmer is trying to narrow down where the program’s behavior deviates from the expected, but this is not the case for novice programmers trying to construct a mental model of the notional machine in the first place. Indeed, the goal of educational programming activities is to develop the viable mental model of program execution that would be required to interpret the results from line-by-line debugging.

Other conceptual models of expression evaluation have been used in other program visualization systems, some of which are closer to the way compiled programs handle expressions, and have appeared in previous systems. For example, the visualization may show a separate evaluation area in which each subexpression is individually evaluated one at a time before the overall result of a full expression or statement is shown (e.g. the Jeliot systems [78], see Figure 7.2). This isolation is undesirable - not necessarily because of a separate evaluation area, but because each subexpression is presented separately and out of context, increasing the cognitive load required for mental tracing. This might be just be annoying for expert users, but it is even more problematic for novices still working to construct a viable mental model. Ultimately, the conceptual models used should be developed in light of how a human mind thinks about a program and the notional machine, not how a computer actually processes them.



Figure 7.1: An example of expression evaluation visualized in Labster. Frames are shown top to bottom, left to right. The next operation to be performed is highlighted or underlined. Light green boxes indicate lvalue-to-rvalue conversions, whereas pink boxes indicate implicit arithmetic conversions.

7.3 Experiment

In order to determine whether Labster was effective in helping students learn about expression evaluation, we conducted a study during the EECS 280 “Arrays and Pointers” lab activities. We used a quasi-experimental, nonequivalent groups design. For a full discussion of our experimental methodology, see chapter 6. Of course, all programming in C++ involves understanding and writing some expressions, but the activities in this lab in particular made use of some particularly complex expressions including the `*`, `&`, `[]`, `++` (prefix), and `++` (postfix) operators.

During the lab activities, students wrote several functions that worked with arrays. In particular, students wrote the following functions:

- A function to traverse an array using indices.
- A function to traverse an array using pointers.
- A function to reverse the contents of an array using pointers.
- A function to shift the contents of an array using pointers.
- (Optional) A function to remove duplicates from an array using pointers.

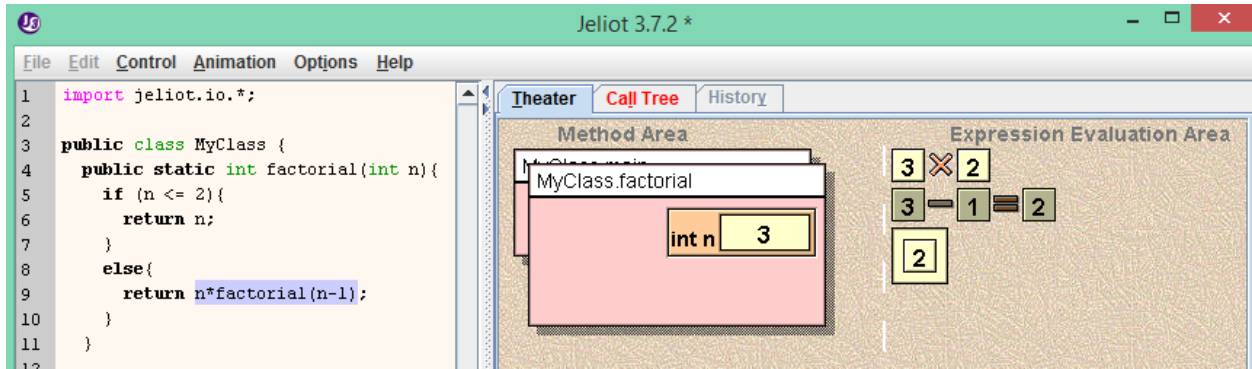


Figure 7.2: A snapshot of the Jeliot 3 program visualization system, which evaluates expressions in a separate area using a graphical representation.

The conceptual questions on the lab surveys involved tracing code arrays and pointers and are shown in Figure 7.3. An understanding of expression evaluation is essential to correctly predict the result of each of the given code snippets.

Assume the following code has been run.

```
int arr[] = {2, 3, 5, 8};
int *ptr = arr;
```

For each code snippet below, indicate which of the possible answers shows the correct contents of the array after the snippet is executed. Consider each snippet separately, starting from the original array each time – do not work cumulatively. (Hint: Work each out on paper first, then select your answer.)

<pre>int arr[] = {2, 3, 5, 8}; int *ptr = arr; arr[3] = arr[2] + 1;</pre>	<pre>int arr[] = {2, 3, 5, 8}; int *ptr = arr; *++ptr = 0; *ptr++ = 1; *ptr = 4;</pre>	<pre>int arr[] = {2, 3, 5, 8}; int *ptr = arr; *(ptr + 2) = 7; (*ptr)++;</pre>	<pre>int arr[] = {2, 3, 5, 8}; int *ptr = arr; ptr = arr + 1; ptr++; *ptr = 0;</pre>
<input type="radio"/> {2, 3, 5, 6} <input type="radio"/> {2, 3, 4, 8} <input type="radio"/> {2, 3, 6, 6} <input type="radio"/> {2, 4, 4, 8}	<input type="radio"/> {0, 3, 4, 8} <input type="radio"/> {0, 1, 4, 8} <input type="radio"/> {2, 1, 4, 8} <input type="radio"/> {2, 0, 4, 8}	<input type="radio"/> {2, 3, 8, 8} <input type="radio"/> {3, 3, 7, 8} <input type="radio"/> {2, 3, 7, 8} <input type="radio"/> {8, 3, 5, 8}	<input type="radio"/> {3, 0, 5, 8} <input type="radio"/> {2, 0, 5, 8} <input type="radio"/> {2, 3, 0, 8} <input type="radio"/> {0, 3, 5, 8}

Figure 7.3: The code-tracing task from the Arrays/Pointers lab surveys.

Students in both the control and experimental groups completed lab exercises that were essentially identical except for the use of Labster. Students in the control group were allowed to use the code editor and compiler of their choice to work through the lab exercise, but our experience suggests very few would have run their code with a debugger or any other additional tool. The lab exercises all include testing code provided to students which allow them to assess whether their solutions are correct. Students in the experimental group used Labster in place of their editor and compiler, and were also encouraged to interact with the visualization of their code throughout the

lab exercises and when testing their code.

Because there was no strict requirement, it is conceivable that some students in the treatment group may have completed the exercises without using the visualization significantly or at all. However, it is also the case that students (in either the control or treatment groups) could complete and pass the lab without ever compiling or running their programs, and a number of students won't attempt the lab assignment at all on any given week. The difference between groups should not be seen as based exactly on what students did, but rather on the nature of the lab assignment with which they were presented.

7.4 Results

Several figures summarize the results from the pre-lab and post-lab surveys for the “Arrays and Pointers” Lab.

- Figure 7.4 shows students' performance on the code tracing questions shown in Figure 7.3.
- Figure 7.5 shows responses to self-evaluation questions.
- Figure 7.6 shows responses to questions about the efficacy of the lab activities.
- Figure 7.7 shows responses from students in the treatment group to questions about using Labster to work with arrays and pointers.

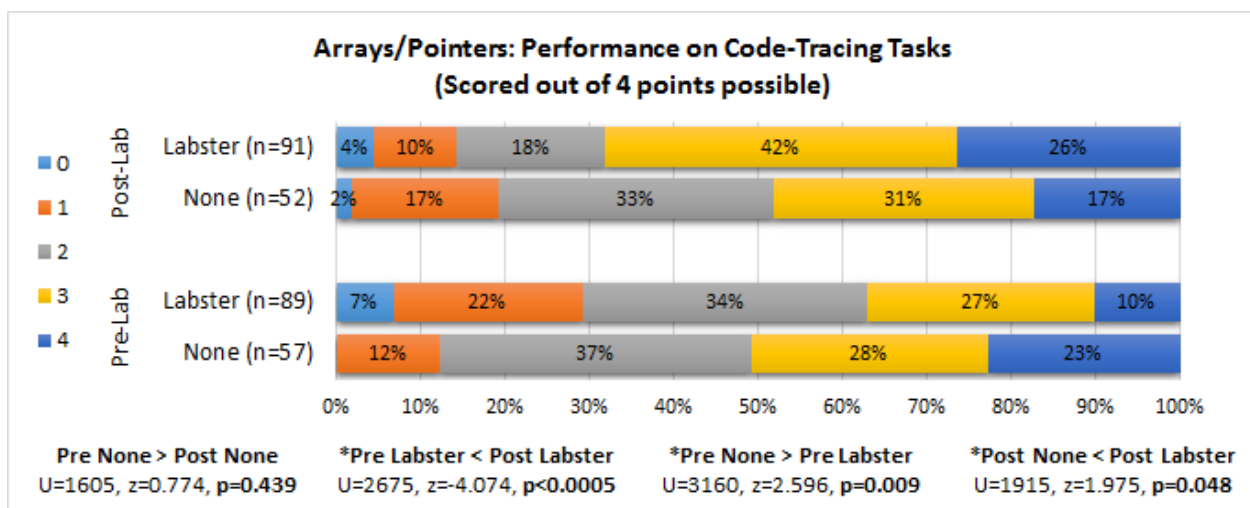


Figure 7.4: Evaluation of students' code-tracing skills before and after the “Arrays and Pointers” lab in EECS 280. Students in the treatment group used Labster, while students in the control group did not use any visualization system (“None”). Significant results are denoted with *.

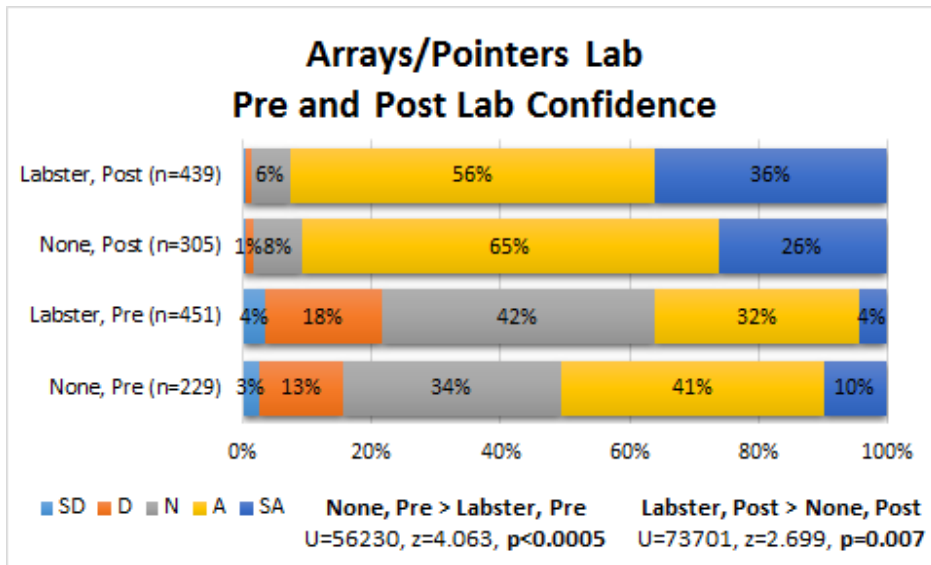


Figure 7.5: Students’ responses to self-assessment questions before and after “Arrays and Pointers” lab in EECS 280. Students in the treatment group used Labster, while students in the control group did not use any visualization system (“None”). Significant results are denoted with *.

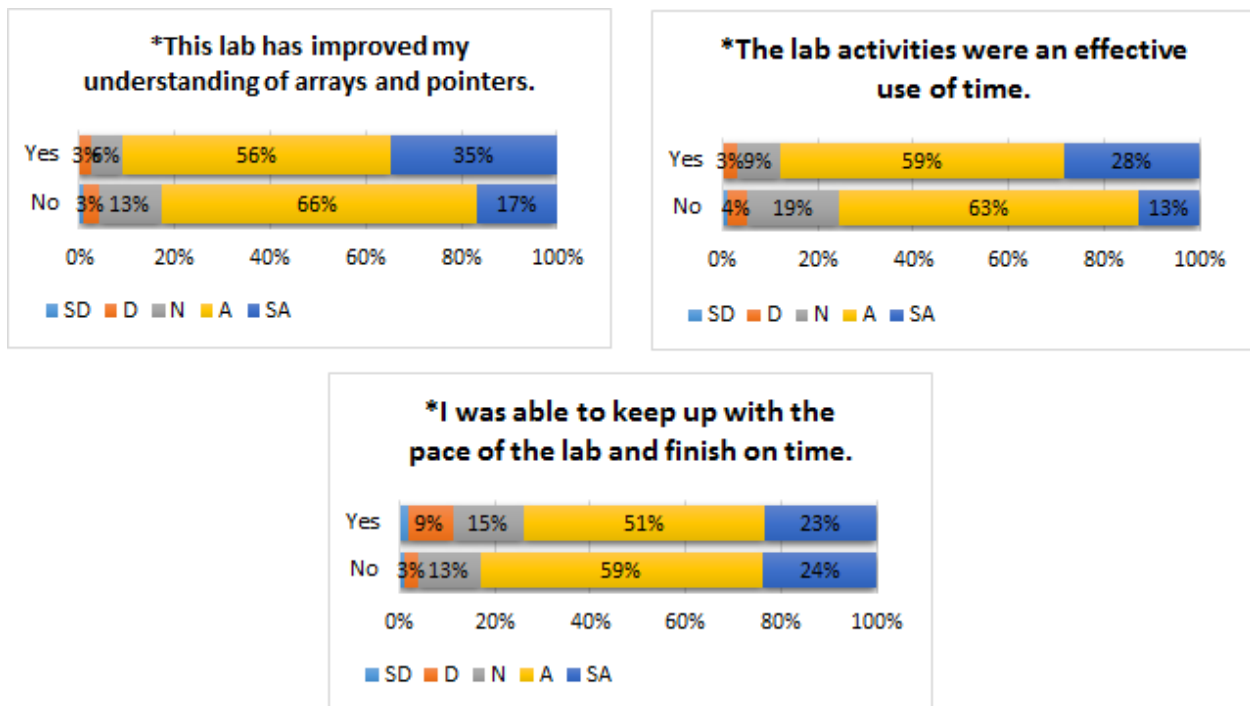


Figure 7.6: Students’ responses to questions about efficacy of the lab exercises before and after “Arrays and Pointers” lab in EECS 280. Students in the treatment group used Labster, while students in the control group did not use any visualization system (“None”). Significant results are denoted with *.

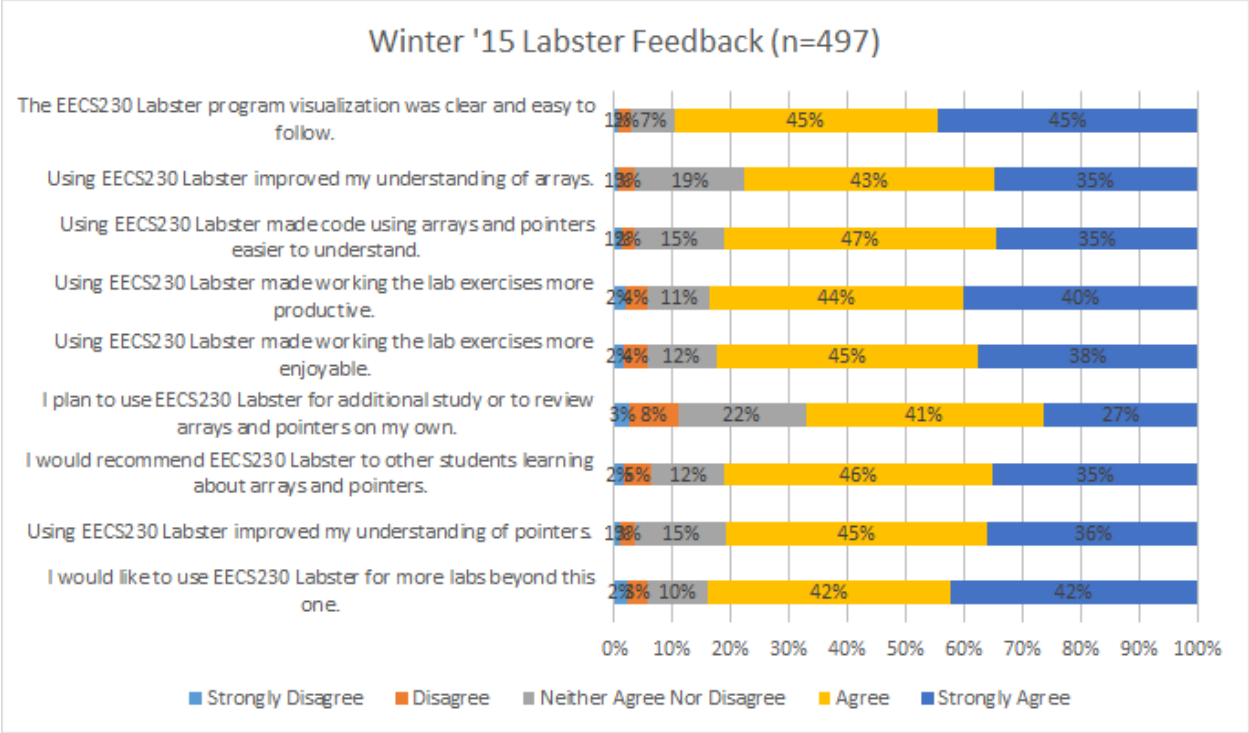


Figure 7.7: Student responses to survey questions after using Labster in the Winter 2015 term. (Treatment group only.)

CHAPTER 8

Experiment: Recursion

Recursion is a fundamental concept in computer science, yet recursive programming is often quite difficult for beginning students. One contributing factor to this difficulty is the fact that in order to use recursion, a student must have a viable understanding of several other underlying concepts. In particular, understanding the way function calls and the call stack work is an essential prerequisite to reading and writing recursive code. In this chapter, we investigate the use of two different conceptual models for recursion that can be presented to students as part of an interactive program visualization system. We compare a novel “code stack” model to the kinds of “static source” models used in many contemporary program visualization systems. Students who used the “code stack” model saw greater improvements in confidence and performance on examinations and also evaluated the lab activities and use of program visualization more positively.

8.1 Background

8.1.1 Mental Models of Recursion

Inexperienced programmers often have non-viable models of recursion [112, 113, 114, 115]. For example, in the “looping” model of recursion, students can follow recursive calls forward but do not account for pending work to be done after each returns. Rather than a stack of function calls, only the most recent one is considered. Such a model is somewhat workable in cases where the final answer is computed at the base case, but not in any where the passive control flow is significant.

Students must develop a mental model that accounts for both active and passive control flow [114] in order to master recursive programming. The “copies” model is generally thought to be the one held by experts, and does implicitly account for passive control flow. Each invocation of the function is seen to create a new copy, and when the function returns the current copy is destroyed and the previous one is resumed. However, students who appear to have a copies model for tracing code may not truly understand recursion, especially when important computation is

done in the passive control flow [116]. The copies model implicitly accounts for passive flow, but it does not strongly emphasize the dynamic processes that take place when calling or returning from individual invocations in a recursive call chain. Students may either need to be explicitly taught about these processes or exposed to them by a visualization that emphasizes these aspects of recursion.

8.2 Control Flow and Subcall Execution in the Notional Machine

George [80] argues that difficulties students have understanding recursion are largely due to an inadequate mental model of subcall execution. Because students don't understand what really happens when a function is called, they don't have the proper foundation required to build an understanding of recursion. George recommends visualizations of code execution should be based on a *dynamic-logical* model of subcall execution, which emphasizes the creation of a separate, unique instance of every called function.

The EROSI (Explicit Representer Of Subprogram Invocations) Tutor [117] is a program visualization tool that animates subcall execution in pre-made examples following George's *dynamic-logical* [80] model. Sequential execution through program text is shown by highlighting the currently executing statement and each function call spawns a new, separate box that displays the program text for that function's execution. The design of the animation is such that both active flow into subcalls and passive flow back to the point of invocation are emphasized.

Experiments with EROSI revealed it was effective in leading students toward development of a viable copies model of recursion [117], which makes sense as a natural consequence of the dynamic-logical conceptual model. The authors also report that incorrect student responses on post-treatment evaluations were often due to exogenous factors rather than to non-viable models of recursion. However, these included misconceptions about variable storage and updating that indicate an incomplete understanding of the way each function has its own separate "instances" of local variables and parameters with the same name. This could rightly be considered an important parallel aspect of a viable copies model of recursion that deals with memory as well as execution. EROSI did not include a memory diagram for these experiments, which perhaps could have ameliorated these issues.

8.3 The *Static Source* Model in Contemporary Systems

Many program visualizations use a variation on what we will call the *static source* model. In these systems, the source code for the program is shown as an immutable whole, and does not change as the program runs. Rather, as different parts of the program run, the visualization provides an indication of which part of the code is executing, such as an arrow pointing to the current line or a highlight showing the currently evaluating expression. When a function is called, the indicator essentially “jumps” to the source code for the called function. Systems with this model often provide some additional information about which functions have been called, such as a stack trace or visual depiction of stack frames for each function invocation in memory. The distinguishing feature of a *static source* model, however, is that a function’s source code itself is not duplicated when that function is called. When a function is called recursively, only the source code for the most recent call can be represented.

Many contemporary visualization systems, including The Online Python Tutor [79], The Teaching Machine [95], and Jeliot 3 [78] use a *static source* model. These systems are shown in Figures 8.1-8.3.

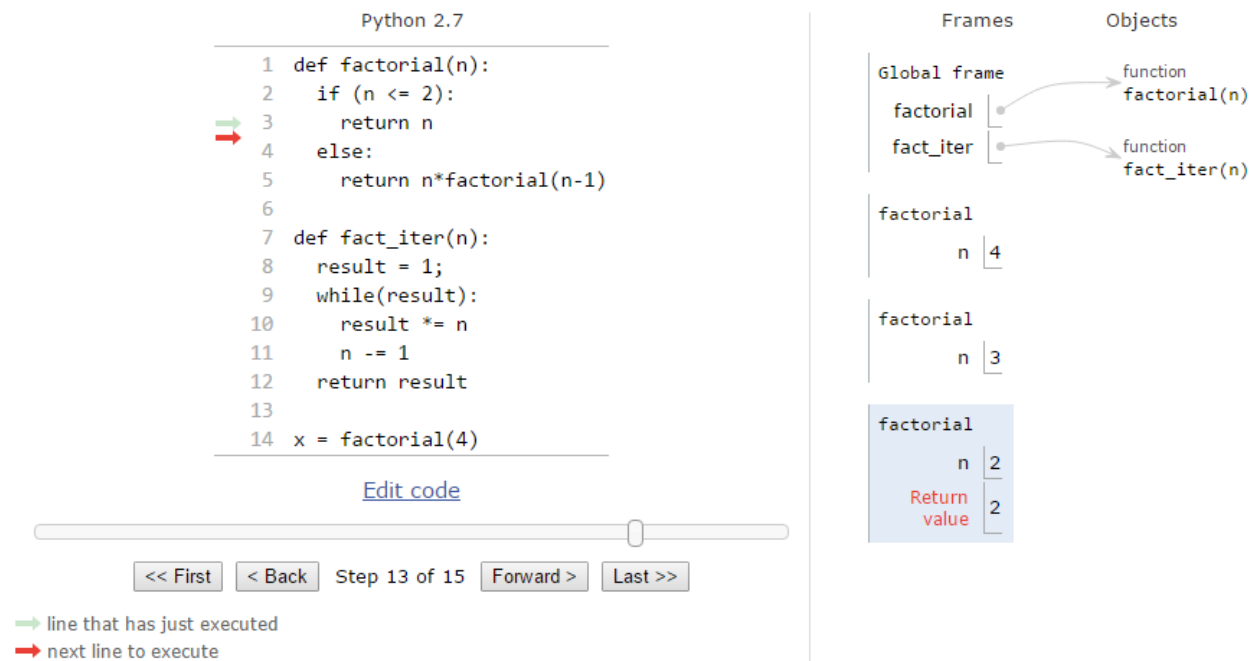


Figure 8.1: A snapshot of the Online Python Tutor visualizing recursive calls in a `factorial` program. The base case has been reached and the value 2 is about to be returned. A *static source* model is used. The user can see the stack frames for each invocation in the memory diagram, but no details of the current point of execution in the previous calls are shown.

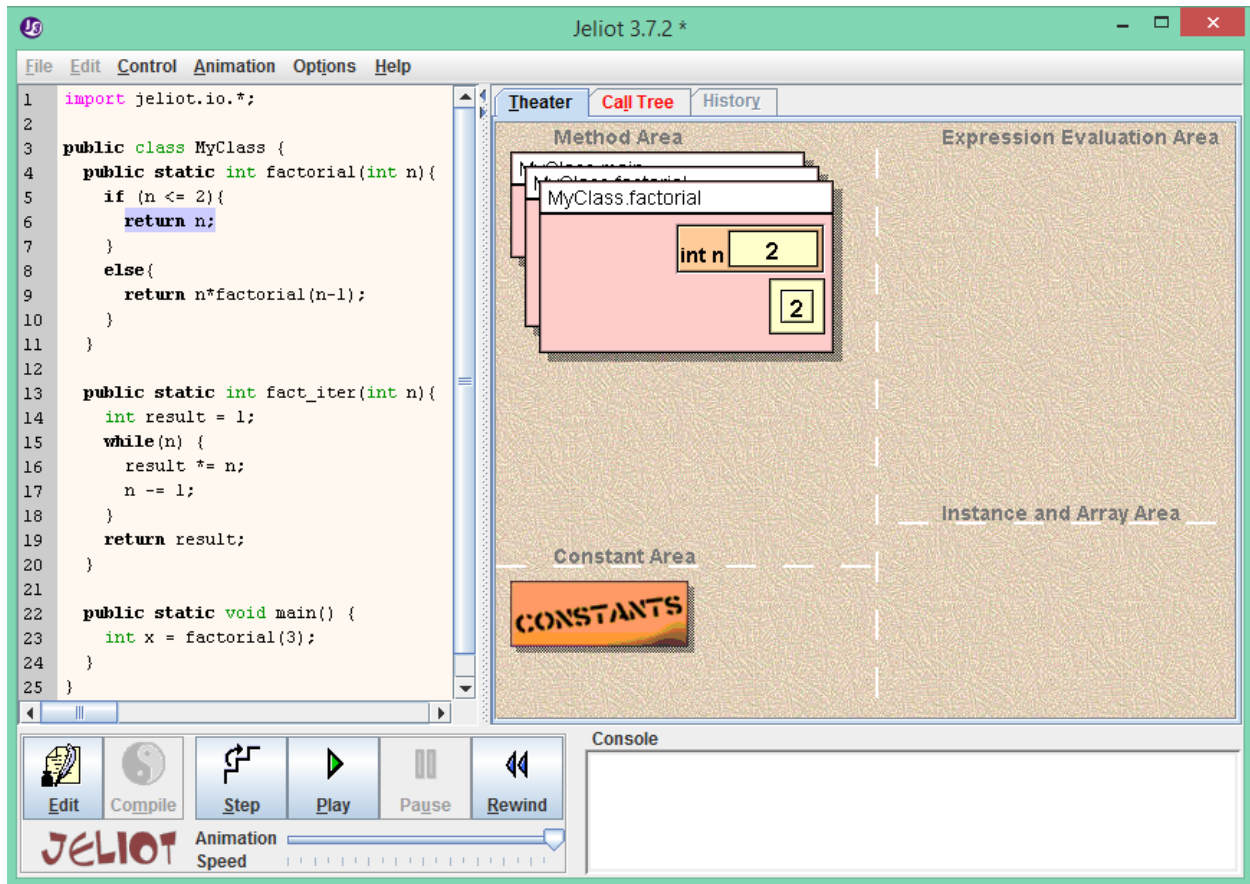


Figure 8.2: A snapshot of Jeliot 3 visualizing recursive calls in a factorial program. The base case has been reached and the value 2 is about to be returned. A *static source* model is used. Although the user can tell how many calls are overlaid in the method area, no details of the previous calls are shown.

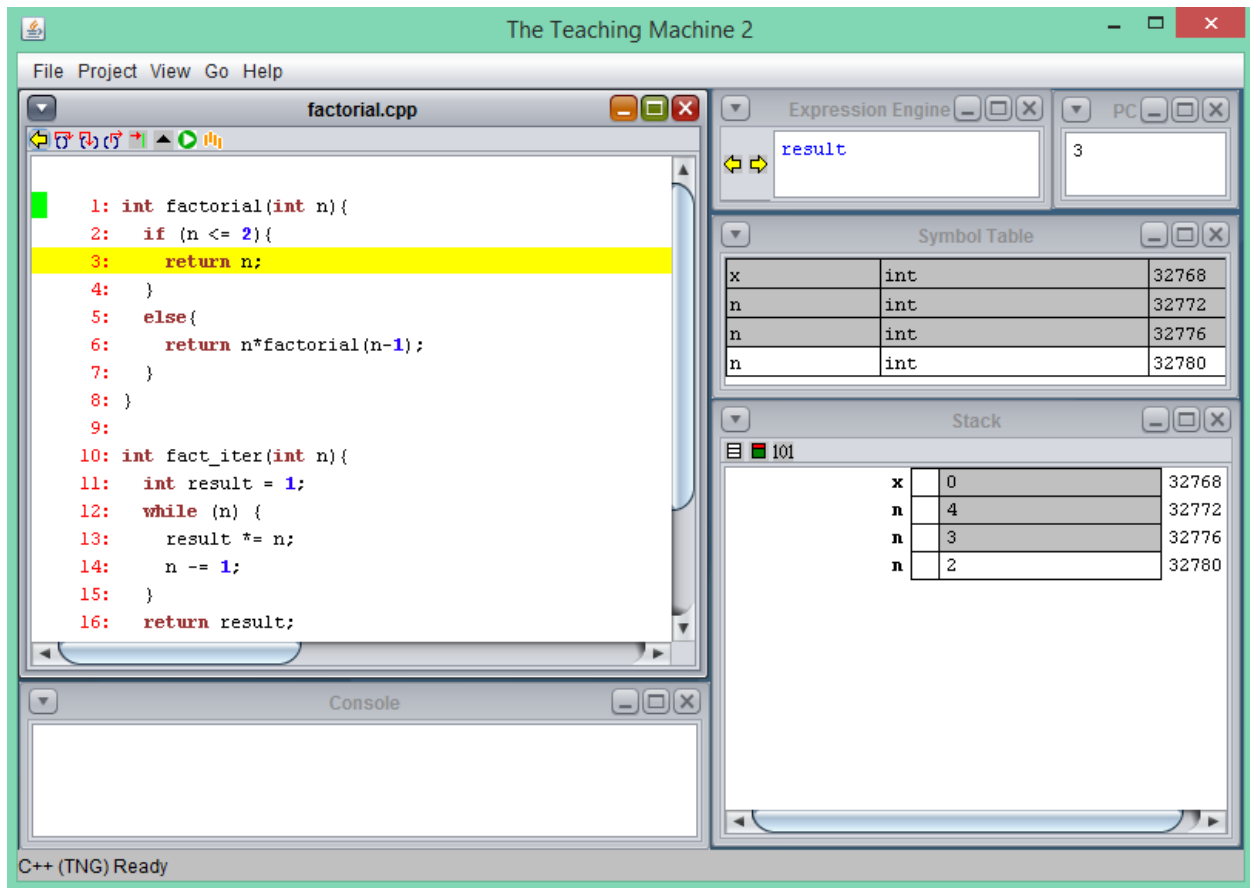


Figure 8.3: A snapshot of The Teaching Machine visualizing recursive calls in a factorial program. The base case has been reached and the value 2 is about to be returned. A *static source* model is used. The user can see the values of each local variable *n* in memory, but no details of the current point of execution in the previous calls are shown.

8.4 The *Code Stack* Model in Labster

The conceptual model of subcall execution presented by Labster draws on George’s *dynamic-logical* model but additionally incorporates the ideas of using the source code itself as context and presenting two parallel stacks for both execution and memory. In Labster, the code being executed is shown distinctly from the source code for the program, and each function call essentially results in another copy of the source code for that function being represented. This is in stark contrast to a *static source* model.

At the beginning of each run through the visualization, the only code showing is that of the `main` function. As the user steps into a function call (active flow), the source code for the called function appears above that for the previous call and a associated stack frame is visualized in the

memory display (see figure 8.4). The calling function is dimmed to indicate it is “on hold” and the location from which the call was made is underlined to indicate where the passive will return to after the most recent call is finished (passive flow). When a return statement is stepped through in Labster, the evaluated value of the returned expression is animated moving back to the location from which the call was made (see figure 8.5).

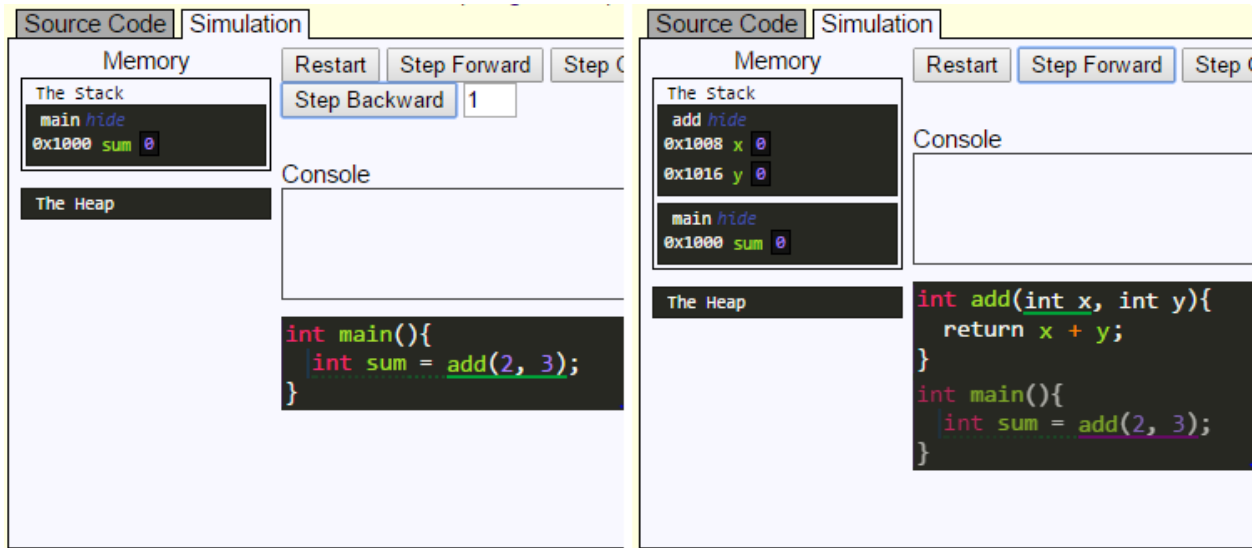


Figure 8.4: Progression from left to right: When a function is called, a new invocation with its own copy of the function code is shown and a new stack frame is displayed in the memory diagram.

Labster emphasizes the passive flow in a number of ways. A function is dimmed after calling another to indicate it is “on hold” and the point to which execution will return is highlighted. When a return does happen, the function is “reactivated” and un-dimmed as the returned value is animated moving from the return statement to the evaluated value of the function call. Additionally, the appropriate context to which a function call will return is always available, because the source code for the previous invocation (including the partially evaluated expression containing the recursive function call!) is always displayed underneath the current invocation.

Labster also visualizes the Tail Call Optimization (TCO) that many compilers perform to reduce the space complexity of tail recursive code. Because a tail call is the last piece of work to be done in a function, the memory used for that function’s stack frame can be reused for the next invocation. Tail recursive calls are shown in a different color in Labster’s visualization, and instead of showing a new, separate function invocation, the old one is simply reused. The values of the arguments to the tail recursive call are animated moving to the parameters of the current function, as if it were literally calling itself. This matches closely the conceptual idea that the only recursive functions that really call themselves (i.e. an invocation calling itself) are those that are

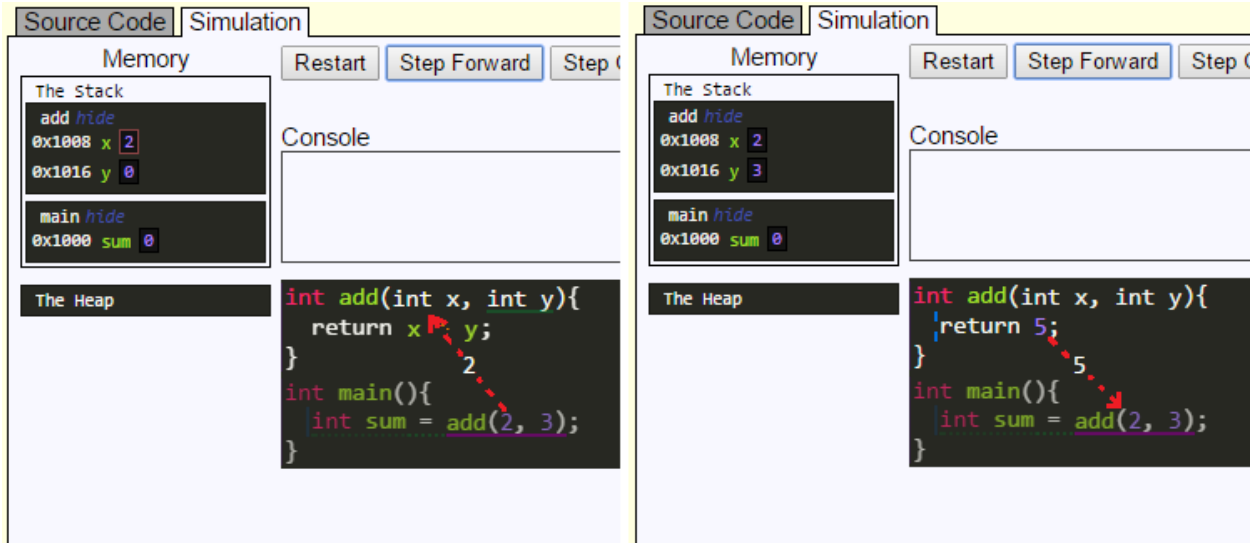


Figure 8.5: Left: An argument is animated moving from its location in the function call expression to a parameter in the new function invocation. Right: When a function returns a value, it is animated moving back to the point from which the function was called. Afterward, the called function and its stack frame disappear before control returns to the calling function.

tail recursive. All other recursive function invocations actually call a new invocation of the same function.

Of course, a major difference between Labster and earlier systems like the EROSI Tutor [117] is that Labster allows students to work with their own code in addition to examples prepared ahead of time by an instructor.

8.4.1 Contrasting the *Code Stack* and *Static Source* Models

The *code stack* model adheres to the general principles of *completeness* and *continuity* [81]. With respect to completeness, one cannot deny that an expert's mental model of the notional machine accounts for separate stack frames of both execution and memory, and it follows that they deserve representation in a conceptual model of the notional machine. This is not to say students are expected to directly encode a visualized representation in their minds, but rather that the visualization needs to provide rich experiences with this aspect of the notional machine from which students can construct their own mental model, whatever the resulting internal representation may be. A visualization that doesn't tell the full story, so to speak, will likely lead to inadequate understanding. A *static source* model does not account for the fact that each invocation of a function

It is also important to view the goal of completeness as showing a complete representation of the notional machine, not the actual machine. Interestingly, the *code stack* model may actually

be more distant from the low-level actuality of how a computer processes subcalls. In compiled languages, at least, the code for a function is translated into line-by-line machine instructions stored sequentially in memory, and there is truly only one instance of these. When a subcall is executed, there is no stack of pending functions waiting to be processed in the passive flow - just a single return address that indicates where to pick up in the source code. A *static source* model matches this quite well, but the goal should be to present an appropriate conceptual model of the notional machine, not the actual machine!

In addition to providing context, showing the entire stack of execution and memory frames is necessary to enable visualization of interactions across different frames. For example, a simple swap function implemented with pointers or references should be visualized in terms of the connection the parameters (i.e. local variables in swap) have to the variables being swapped that exist on a lower frame in the stack.

As the amount of source code at hand grows, a *static source* approach may become more difficult to use. The sheer volume of code to be displayed in the visualization may be overwhelming, whereas a code-stack approach only shows code that is currently being used. Additionally, execution may "jump" from one place in the source code to another, which can be disconcerting for students.

The *code stack* approach also provides a natural way to visualize code for functions included from libraries or implicitly defined by the compiler should be displayed (e.g. an implicitly defined copy constructor in C++). While a *static source* approach could accommodate these sorts of things as a special case, any strategy that involves code dynamically appearing when a function is called is essentially just borrowing from the *code stack* approach.

8.5 Experiment

In order to compare the effectiveness of the *static source* and *code stack* models, we conducted an experimental intervention using a between-subjects, independent groups design during the "Recursion" lab activities for the Fall 2015 term of EECS 280. Students in the course were randomly assigned to two groups and had access to one of two different versions of Labster that differed only in whether a *static source* or *code stack* was used model. For a full discussion of our experimental methodology, see chapter 6.

During the lab activities, students wrote several functions that used iteration, recursion, and tail recursion. In particular, students wrote the following functions:

- An iterative function to print numbers from a recursively defined sequence.

- A recursive (trivially tail recursive) function to print numbers from a recursively defined sequence.
- An iterative function to count the occurrences of a digit in the base ten representation of an integer.
- A recursive function to count the occurrences of a digit in the base ten representation of an integer.
- A tail recursive function to count the occurrences of a digit in the base ten representation of an integer.

Some of the problems given to students on the pre-lab and post-lab surveys required students to write a code snippet or function implementation to perform a particular task. These questions were scored according to a rubric which allowed partial credit for each conceptual piece of a working solution. For the Recursion lab, the problems also specified whether the solution should be written using iteration, recursion, or tail recursion. Responses that used the wrong strategy were considered incorrect even if the code worked otherwise. We did not deduct points for poor formatting or superfluous syntax errors. The two code-writing problems are shown in Figures 8.6 and 8.7.

Write an implementation for a recursive function that determines whether a number is a power of 2.
 Use the following strategy:

The number 1 is by definition a power of 2 (i.e. 2^0).

A number x is a power of 2 if and only if x is even AND $x/2$ is a power of 2.

- You may assume we are working with only positive numbers.
- Your solution does not need to be tail recursive.
- Recall that an integer x is even if and only if $x \% 2 == 0$

Figure 8.6: Code-Writing Task 1: Students are asked to use recursion to write a function that checks for a power of 2.

Any number larger than one that is not evenly divisible by any other numbers than itself and 1 is called prime. Here are the first twenty integers with the primes underlined.

1 2 3 4 5 6 7 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Assume you are given a function called `isPrime` that returns true if its parameter is a prime number and false otherwise. For example:

```
isPrime(3) returns true
isPrime(9) returns false
isPrime(17) returns true
```

Given `isPrime`, we could write a function that counts the number of prime numbers from 2 to `n`. For example, here is an **iterative** implementation of such a function.

```
int countPrimes(int n){
    int total = 0;
    for(int i = 2; i <= n; ++i){
        if (isPrime(i)) { ++total; }
    }
    return total;
}
```

Your task is to write a **recursive** (but not tail recursive) version of this function.

```
int countPrimes(int n){
    // Your code here
}
```

Figure 8.7: Code-Writing Task 2: Students are asked to convert between strategies for implementing a function to count prime numbers.

We also compared the performance of both groups on a recursion question included in the midterm exam, which is shown in figure 8.14. The problem asked students to write a recursive function to filter elements from a tree. This is a much more difficult problem than those on the post-lab surveys, and is also dissimilar from the functions students wrote in the lab exercise. In fact, the lab exercises didn't involve trees at all, but were aimed at teaching students the underlying concepts that are relied on to work with trees recursively. This means that the exam problem likely tests transfer of knowledge more effectively than the problems on the post-lab survey, which were quite similar to the ones students saw during the lab activities.

8.5.1 Labster Blue and Labster Maize

Students were randomly assigned into groups that used either the “Blue” or “Maize” versions of Labster. Labster Blue was an adapted version that used a *static source* model similar to what is used in many contemporary program visualization systems. Labster Maize was the regular version of Labster, which uses the *code stack* model.

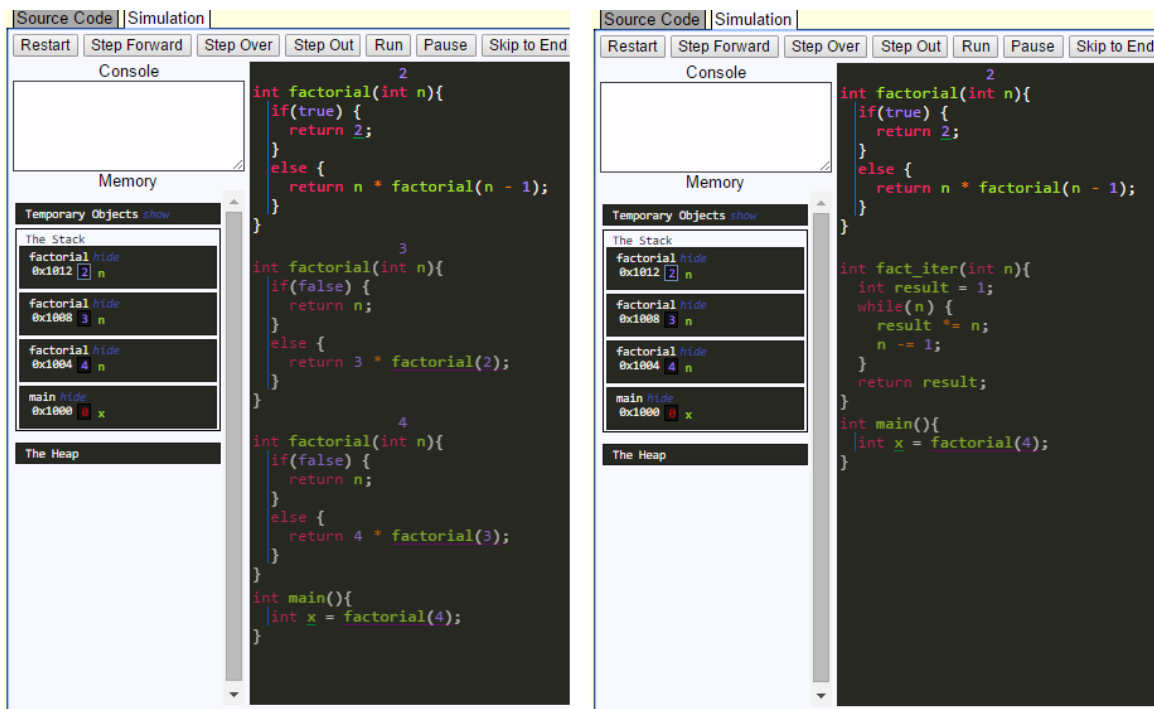


Figure 8.8: A snapshot of the *Maize* and *Blue* versions of Labster visualizing different conceptual models of subcall execution for a factorial program. The base case has been reached and the value 2 is about to be returned. The *Maize* version (left) presents the *code stack* model and displays the source code for functions that have been called in a stack of execution frames, which appear and disappear with function calls and returns. The top frame corresponds to the function that is currently executing. The *Blue* version (right) presents a *static source* model and shows the source code verbatim. The organization of the code is fixed as in the source, but the current function is highlighted. Blue does not show separate invocations of a single function separately.

The two versions are shown side by side in Figure 8.8, which shows a snapshot of the visualization for a factorial program. In Labster Blue, which presents a *static source* conceptual model, only one version of the code for factorial is visible. Even though it has been invoked three times, only the most recent invocation is shown. The purple underline in `main` indicates that line is waiting for a computation to finish, there is no indication of how many calls of `factorial` need to return before that, except for the number of stack frames shown in the memory diagram. Finally, the code for the `fact_iter` function is also shown since it appears in the source code, even though it has not currently been called.

On the other hand, Labster Maize uses the *code stack* model. Instead of displaying code as it appears in the source, a "code stack" is used and each subcall is displayed as a separate invocation. Each individual call to factorial is visualized, and invocations that are "paused" are dimmed. The precise way in which passive control flow works as the stack unwinds is naturally visualized, and the numbers which will be multiplied into the result during the passive flow are evident. The code for the `fact_iter` function is not shown, since it has never been called.

8.6 Results

Several figures summarize the results from this experiment.

- Figure 8.9 shows students' performance on the code writing question shown in Figure 8.6.
- Figure 8.10 shows students' performance on the code writing question shown in Figure 8.7.
- Figure 8.11 shows responses to self-evaluation questions.
- Figure 8.12 shows responses to questions about the efficacy of the lab activities.
- Figure 8.13 shows responses from students in the treatment group to questions about using Labster.

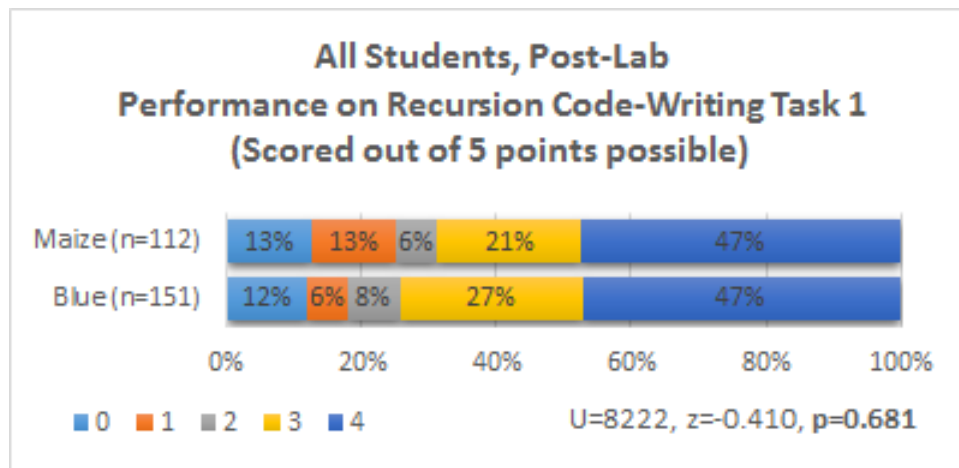


Figure 8.9: Students' performance on code-writing task 1 on the post-lab survey after working with Labster Blue/Maize for the "Recursion" lab in EECS 280. Students in the Blue group worked with a *static source* model in Labster, while students in the Maize group worked with the *code stack* model. There was no significant difference between the groups.

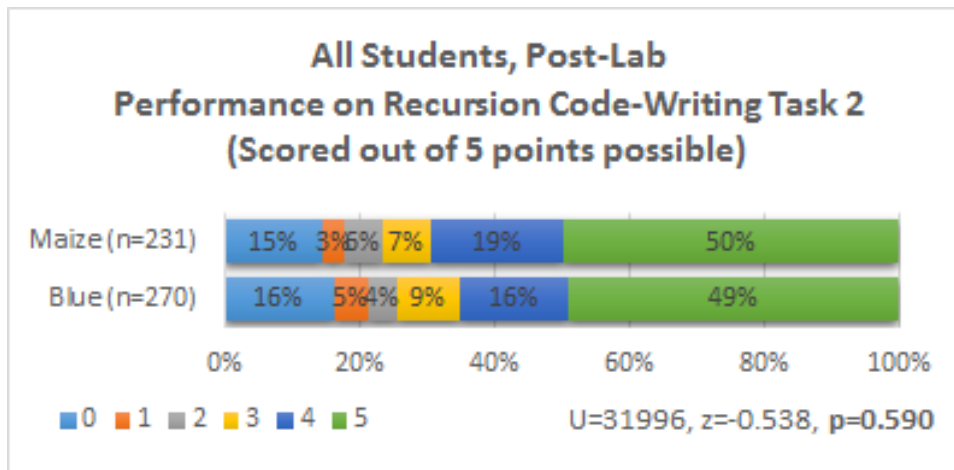


Figure 8.10: Students’ performance on code-writing task 2 on the post-lab survey after working with Labster Blue/Maize for the “Recursion” lab in EECS 280. There was no significant difference between the groups.

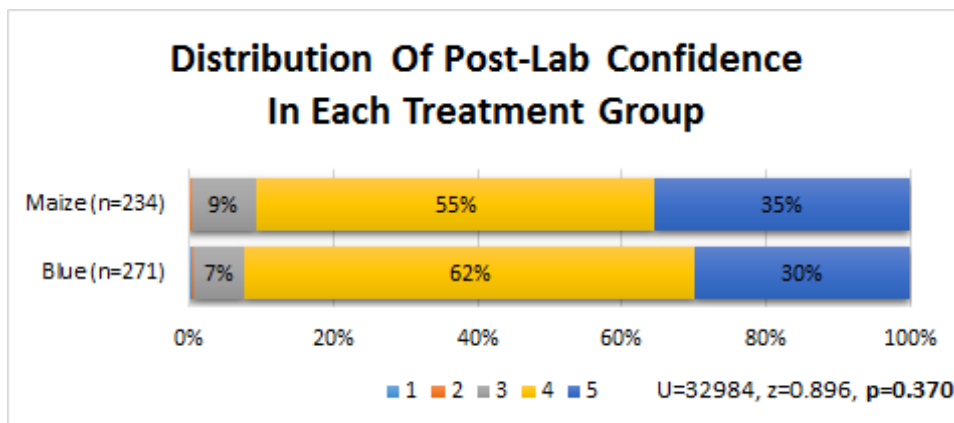


Figure 8.11: Students’ confidence level after working with Labster Blue/Maize for the “Recursion” lab in EECS 280. There was no significant difference between the groups.

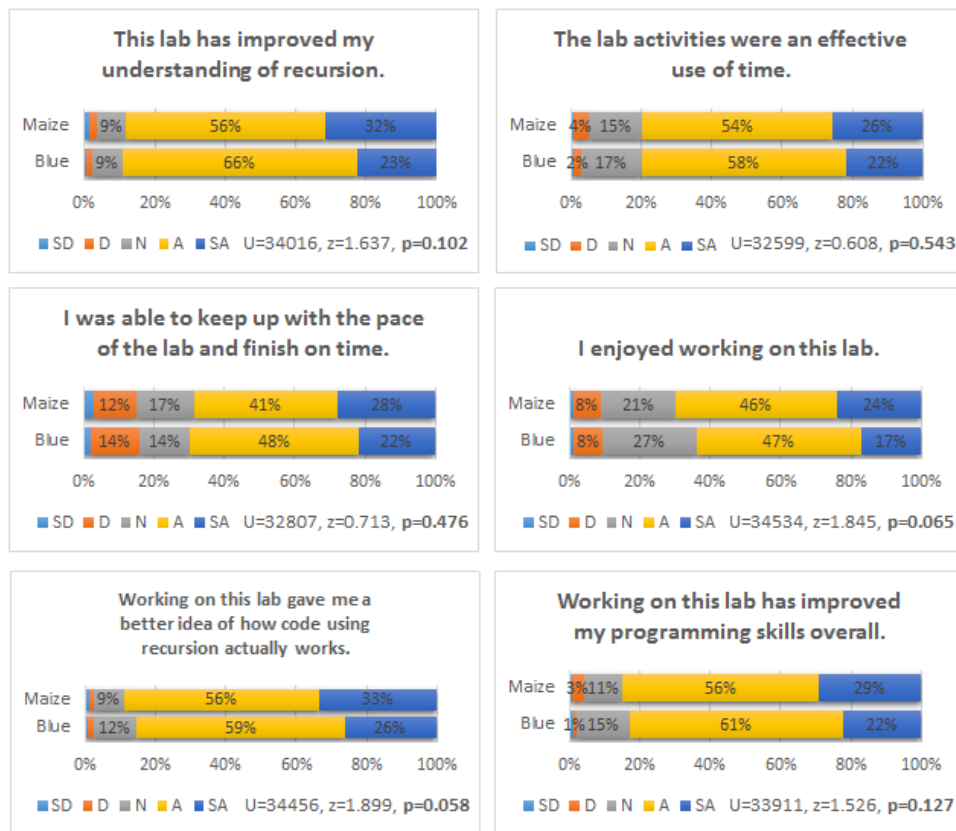


Figure 8.12: Students’ responses to questions about efficacy of the lab exercises after working with Labster Blue/Maize for the “Recursion” lab in EECS 280. Significant results are denoted with *.

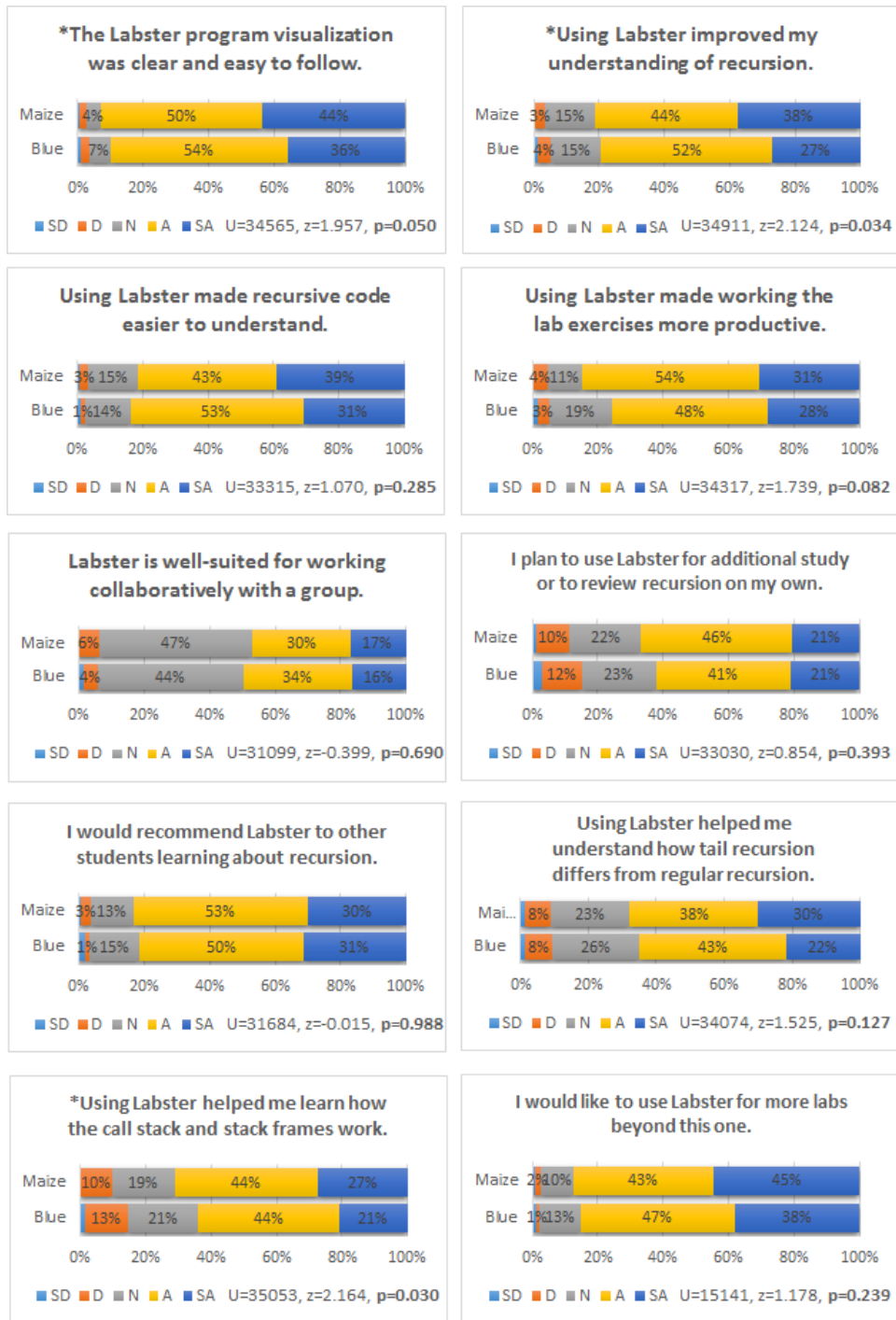


Figure 8.13: Student responses to survey questions after using Labster after working with Labster Blue/Maize for the “Recursion” lab in EECS 280. Significant results are denoted with *.

`tree_filter` is a function that takes a sorted binary tree and a predicate function pointer as input, and produces a list of sorted integers that when passed into the predicate return true. **You may use any of the list or tree functions on the last page of this exam (which you may rip off). You may NOT use any functions other than those listed on the last two pages. Using any other project 2 functions will award you 0 points on this question.**

4a) (15 points) Implement the function `tree_filter()`.

```
// REQUIRES: tree is a sorted binary tree
// EFFECTS: Returns the sorted list of elements which returned true
//           for pred
list_t tree_filter(tree_t tree, bool (*pred)(int)) {
```

Figure 8.14: EECS 280 Fall 2015 midterm question 4a. This question heavily relied on an understanding recursion, but was not directly similar to problems in the lab exercises. We compared scores for students who had spent 1.5-2.5 hours using Labster for lab 2 (Blue n=102, Maize n=73). Blue average: 6.1/15, Maize average: 7.8/15. The difference is significant at $U=2549$, $z=2.108$, $p=0.035$. The difference was more pronounced for students who came into lab 2 with a confidence level of 4 (Blue n=49, Maize n=35). Blue average: 5.8/15, Maize average: 8.4/15. The difference is significant at $U=515.5$, $z=2.046$, $p=0.040$.

CHAPTER 9

Experiment: Navigation Controls

It is well established that actively engaging learners is a critical component of effective program visualization. In particular, many contemporary program visualization systems are designed to support learner engagement at the *controlled viewing* level. Considerable evidence supports putting control of the visualization in students' hands. Intuitively, it encourages them to take a much more active role in visualizing a program and allows them to moderate the flow of information.

However, within the realm of *controlled viewing*, does it matter what particular controls students have at their disposal for navigating through the execution of a program? In this chapter, we present an initial investigation into the impact of the navigation controls provided by a system and how they affect student learning. We report results from an empirical evaluation of two different versions of Labster, one with very basic controls and another with full featured navigation. Students used the system during the “Recursion” lab. We found that for students who were moderately confident in the material before the lab, having more navigation features led to significantly better learning outcomes than those with only basic navigation controls. In general, students evaluated the system and its usefulness much more favorably if full navigation features were available and also spent more time working with the system.

9.1 The Role of Navigation in Program Visualization

The mechanisms and controls provided for navigating through the execution of a program are an essential component in the design of interactive program visualization systems. Navigation controls determine a wide array of characteristics for a system, including the granularity with which students are able to move through a program, whether they are able to run it automatically to completion, whether they are able to navigate backward as well as forward, how easily they can navigate to an arbitrary part of the code, etc.

Our focus in this chapter is on students engaging at the *controlled viewing* level of direct engagement and at the *modified content* and *own content* levels of content ownership. However, the

controlled viewing mode of engagement also serves as the foundation on which higher levels of engagement are built, and our work largely applies to those as well. For example, exercises that have a student *respond* to questions about a visualization may require the student to move back and forth through particular steps of the visualization in order to determine the answer. In another case, a student who is *presenting* a visualization to others will almost certainly want to control the execution of the visualization as well.

A benefit of sophisticated navigation is that it allows students to process a visualization at whatever level of granularity they desire. A system which only allows navigation at a fixed level of granularity is prohibitive to students' ability to learn from visualization. If the granularity is too coarse, the visualization lacks *completeness* [81] and students may not be able to see all the details relevant to whatever they are studying. On the other hand, if a visualization is fixed at a very fine level of granularity, students may be overwhelmed by unnecessary details. This is particularly noticeable if a program contains subroutines which perform a conceptually simple task (considered in the abstract), but whose implementation details are not relevant to a student's concerns.

Navigation controls also affect which patterns of use by students are convenient. This plays into a system's ability to effectively support various levels of content ownership. When students work with a fixed example, the pattern of use is likely fairly linear. A student is likely to work through the execution of the code in-order, perhaps pausing to step backward a few times during complicated pieces, but mostly moving incrementally forward. After the concept is initially grasped, a student may want to restart the visualization and run through it once more to make sure they understand what is going on.

As the level of content ownership increases toward students working with their own code, the pattern of interaction shifts away from a linear progression through a fixed example to an iterative process of modifying and visualizing code. Each time the code is modified, a student most likely only wants to visualize a very particular part of their program in order to see the direct effects of the change they made. Navigation through the code is decidedly non-linear in this case, and certain parts of execution may be visualized much more than others. Efficient navigation tools are essential to support this kind of interaction. If students are not quickly able to navigate to the particular point of execution they are interested in, they may be much less likely to use the visualization at all. Few students would put up with the tedium of stepping through the program line-by-line from the beginning and instead will avoid using the visualization.

9.2 Navigation Features in Existing Systems

Existing visualization systems that support engagement at the *controlled viewing* level with *own content* are too numerous to discuss individually here, but a comprehensive review can be found in

[18] sections 5.2-5.4.

All systems that allow *controlled viewing* must support basic navigation features like moving one step forward, but systems vary by whether they only allow stepping line-by-line or by fine grained expression evaluation. An additional feature of some systems (e.g. The Teaching Machine [95], Online Python Tutor [79], WinHIPE [72], VILLE [94], and many others) is the ability to step backward as well as forward through a program’s execution. The review given in [18] indicates which systems allow expression-level navigation and whether backward stepping is possible.

The ZStep 95 system [108] visualizes the execution of Lisp programs and provides a wealth of different navigation options. Users are able to step both forward and backward in single increments or automatically step however far is needed to reach the first state that is graphically different from the current one with respect to the visualization. Additionally, clicking on an expression in the program display will automatically run either forward or backward to the point at which that expression is executed. These navigation features were evaluated heuristically for ZStep 95, but not formally.

Many reversible debugging tools exist [118], but these generally do not incorporate a visualization and the focus of such tools is often on scalability and use by experts for complex programs rather than use for educational purposes.

9.3 Varied Navigation Controls in Labster

We investigated the impact of navigation controls by comparing groups of students who used two different versions of the Labster system, which we will call the “Old” and “New” versions for lack of better terms (see Figure 9.1). The “Old” version only provided three navigation buttons: step forward, step backward, and restart. Additionally, the user could use a mouse wheel to scroll forward or backward through several steps. The “New” Labster was an updated version of the first and added several navigation features including buttons to step over or step out of a function call or other language construct like a loop as well as buttons to automatically run forward at varying speeds until the pause button is clicked. Finally, the “New” system allows users to simply click on a statement in the code display and the visualization will automatically run either forward or backward to the beginning of that statement’s execution.

There are also a few other differences between the Old and New versions of Labster, including different color schemes and that the stack grows upward in one version and downward in another. However, these changes should be trivial compared with the major overhaul of navigation functionality.

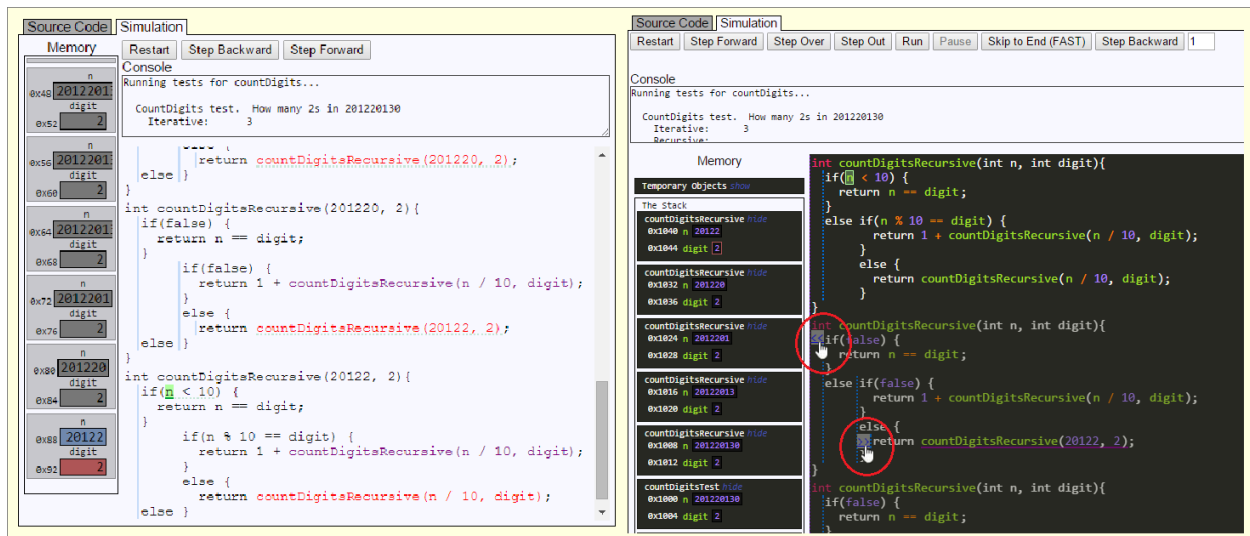


Figure 9.1: A screen capture of both versions of Labster simulating code from the Recursion lab exercises. The “Old” version (left) only allows the user to step forward/backward one step at a time using the buttons or by scrolling the mousewheel. The “New” version (right) offers these features plus several additional buttons for navigation, including a way to automatically run the program at varying speeds. Additionally, hovering the mouse over a statement reveals a link (red circles) that will immediately run the visualization forward or backward to that point in the code, depending on whether it has already executed or not.

9.4 Experiment

We conducted a quasi-experimental intervention during the EECS 280 “Recursion” lab in which each group used a different version of Labster. Students during the Fall 2014 term used the “Old” version of Labster and students during the Winter 2015 term used the “New” version. We used a nonequivalent groups design. To ensure meaningful comparisons, we used pre and post measures of each group and we conduct our analyses accordingly. Between terms, the lab exercises were identical except for the version of Labster that was used. For a full discussion of our experimental methodology, see chapter 6.

During the lab activities, students wrote several functions that used iteration, recursion, and tail recursion. In particular, students wrote the following functions:

- An iterative function to print numbers from a recursively defined sequence.
- A recursive (trivially tail recursive) function to print numbers from a recursively defined sequence.
- An iterative function to count the occurrences of a digit in the base ten representation of an integer.

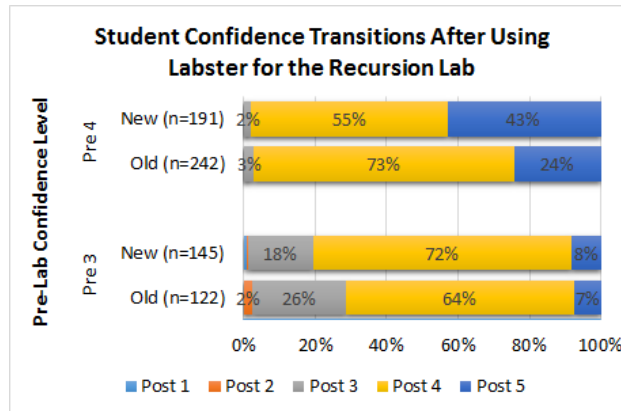


Figure 9.2: Distribution of confidence scores from post-test responses, grouped according to pre-test confidence and treatment condition.

- A recursive function to count the occurrences of a digit in the base ten representation of an integer.
- A tail recursive function to count the occurrences of a digit in the base ten representation of an integer.

Some of the problems given to students on the pre-lab and post-lab surveys required students to write a code snippet or function implementation to perform a particular task. These questions were scored according to a rubric which allowed partial credit for each conceptual piece of a working solution. For the Recursion lab, the problems also specified whether the solution should be written using iteration, recursion, or tail recursion. Responses that used the wrong strategy were considered incorrect even if the code worked otherwise. We did not deduct points for poor formatting or superfluous syntax errors. The two code-writing problems are the same as those shown in Figures 8.6 and 8.7 from chapter 8.

9.5 Results

9.5.1 Post-Lab Confidence

Figure 9.2 shows the post-lab confidence distributions for students from pre-lab confidence groups 3 and 4 according to whether students used the Old or New version of Labster. For group 3, the difference in post-lab confidence distributions was not significant ($U=9675.5$, $z=1.617$, $p=.106$). For group 4, students who had used the New version of Labster had significantly higher post-lab confidence than those who had used the Old version ($U=27414$, $z=4.00$, $p<0.0005$).

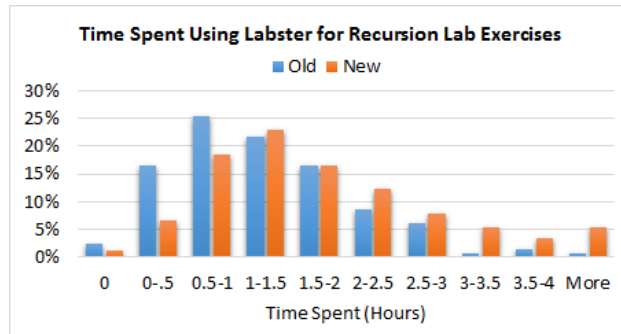


Figure 9.3: Students who used the New version of Labster tended to spend longer working with the system during the Recursion lab week.

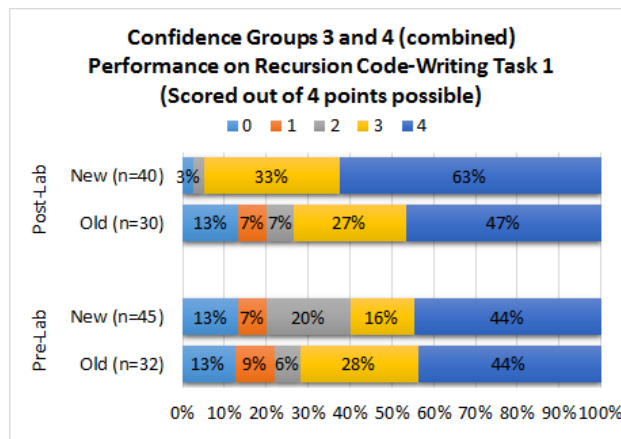


Figure 9.4: Pre-test scores for code-writing task 1 were not significantly different, but post-test scores for students who had used the New version of Labster were nearly significantly better ($U=740$, $z=1.858$, $p=.063$).

9.5.2 Post-Test Performance

Students were tested on a number of concept-based and code-reading questions (e.g. “does this recursive function have a base case?”, “what does this function output?”, etc), but there were no appreciable differences in post-test scores for any of our groups. It is not surprising that visualization may not be as impactful for these kinds of questions. Additionally, one of the code-tracing questions turned out to be too easy and a vast majority of students got it right on both the pre and post tests.

Figures 9.4, 9.5, and 9.6 show performance for students from the 3 and 4 confidence groups on post-test code-writing problems. Group 4 seemed to benefit the most from using the New version. It is less clear that there was a difference between the Old and New versions for group 3, perhaps because they were less poised to take advantage of the additional navigation features than the more proficient students. A common trend for each of the code-writing questions is that the number of zero scores is much reduced for students who used the New version, meaning fewer students were

failing to apply the correct strategy.

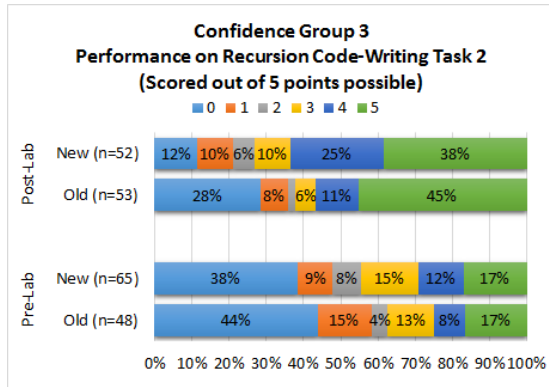


Figure 9.5: Confidence group 3 students in both terms scored similarly on code-writing task 2 on the pre-test and the post test, and the scores on the post-test were not significantly different ($U=1454$, $z=.510$, $p=.610$).

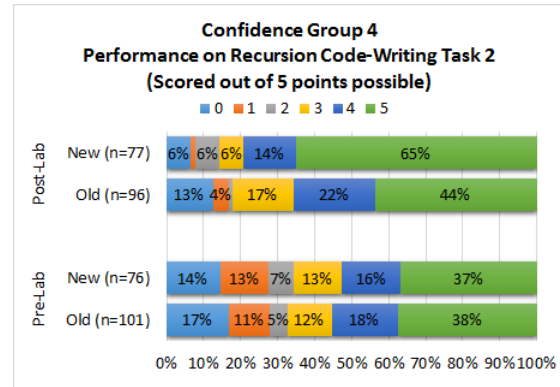


Figure 9.6: Confidence group 4 students in both terms scored similarly on code-writing task 2 on the pre-test, but post-test scores for students who had used the New version of Labster were significantly better ($U=4480$, $z=2.612$, $p=.009$).

9.5.3 Labster Evaluation and Time Spent

The first three survey questions in Figure 9.7 essentially ask students whether using Labster was beneficial for their learning. Students who used the New version of Labster felt much more strongly that it had a positive impact, and this is in accordance with improved performance on code-writing questions. The contrast between whether students felt using Labster made the lab exercises more productive is particularly stark, perhaps because students found moving only one step at a time in the Old version to be tedious. Students who used the New version were also more likely to indicate they planned to use the tool again and that they would recommend it to other students learning about recursion.

Figure 9.3 shows how much time students spent working with Labster. It turns out that students using the New version of Labster tended to spend more time working with the system than those with the Old version. Because students also evaluated the productivity of the New version more favorably, a potential explanation for the discrepancy is that students may have become frustrated with the Old version and not worked through the lab as completely.

9.6 Summary

We have shown that the navigation controls provided for engagement with a program visualization system at the *controlled viewing* level can significantly affect students' confidence, code-writing

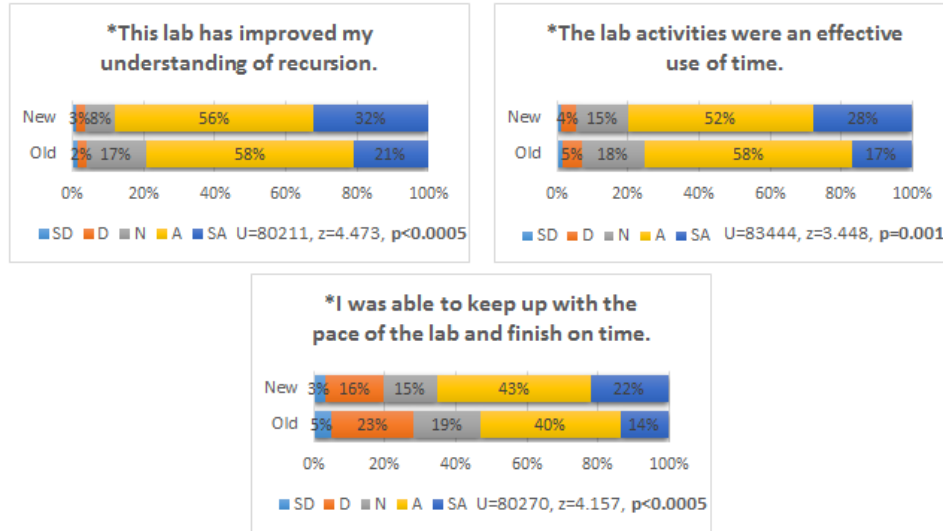


Figure 9.7: Post-lab survey results. Students who had used the New version of Labster found the lab activities more effective. All results shown are statistically significant. Old n=443, New n=429.

proficiency, time spent working with the system, and evaluations of the system and activity in which they participated. In this work, we found that a full set of navigation controls is superior to a very basic set, but future research may take a more fine-grained approach to determine precisely the ways in which students actually use the navigation tools given to them. Finally, innovative navigation features like clicking part of the code to run to that location must be more fully explored and evaluated.

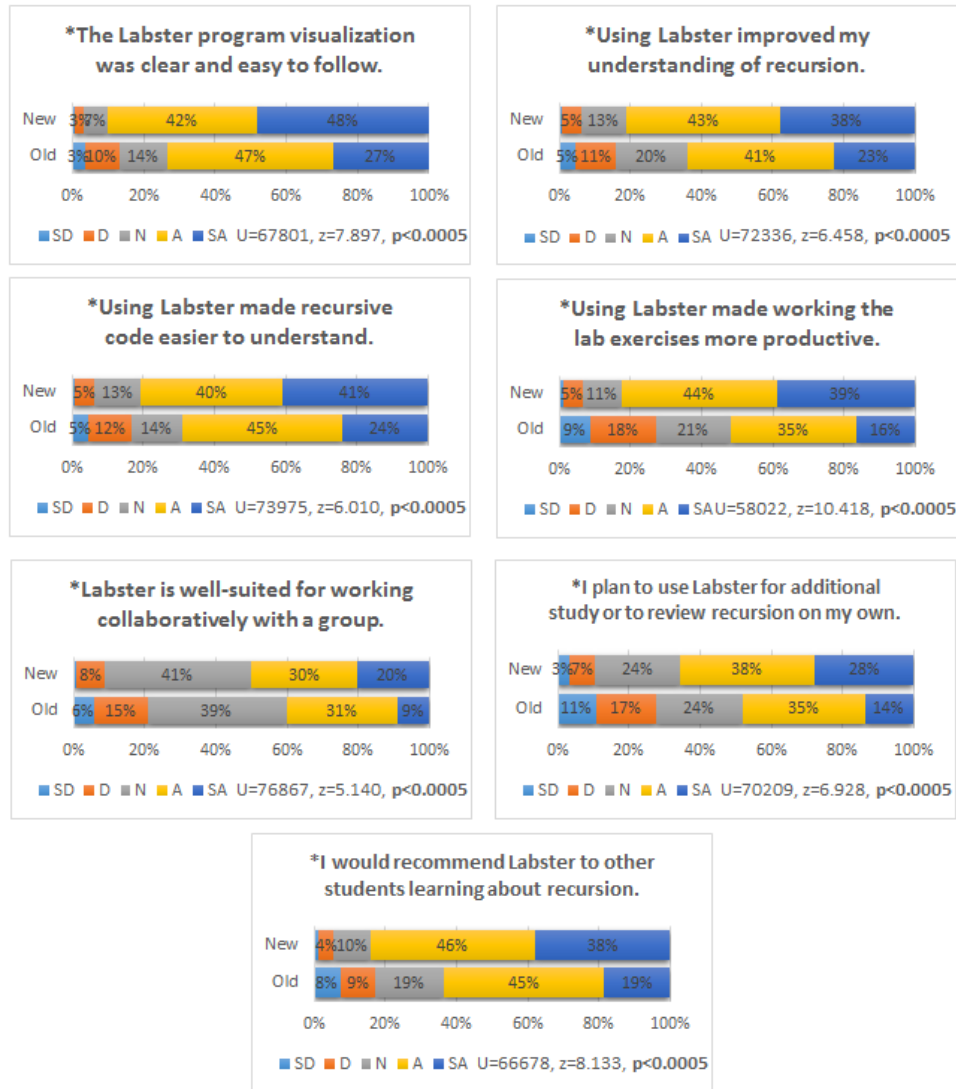


Figure 9.8: Post-lab survey results. Students who had used the New version of Labster evaluated the system more favorably and felt more strongly that it was beneficial for their learning. Old n=443, New n=429.

CHAPTER 10

Discussion

In this chapter, we further discuss our individual experiments, offer interpretations of the results, and identify general trends.

10.1 What Kinds of Learning Outcomes are Affected?

Across our experiments, a number of trends emerged concerning the types of learning outcomes that were most sensitive to our interventions. We found significant differences in individual experiments measuring students' ability to read, understand, and write code, especially when the questions students were asked involved a higher amount of transfer knowledge. Students' confidence and evaluation of the effectiveness of the lab activities were also affected in some cases, depending on whether Labster was used and which conceptual models were presented. However, measurements of students' performance on tests of conceptual recall and static analysis of patterns in source code did not reveal significant differences. In this section, we discuss generally where significant results were and were not found at a high level.

10.1.1 Conceptual Recall

Throughout our experiments, the use of Labster did not significantly impact students' performance on tasks that tested their recall of conceptual material related to lab topics. For example, we found no significant differences in students' performance on questions about properties of arrays across different interventions. This is not surprising, because the lab exercises did not target this kind of learning at all, and even improvements between pre and post lab tests were quite small. This does not necessarily mean that program visualization is ineffective for improving students understanding of general programming concepts, but a more targeted selection of conceptual questions would likely be needed to expose such an effect. For example, a question about the nature of references or of value semantics might well be affected by the use of the visualization, because a concrete

conceptual model grounds these concepts and provides students with experiences that illustrate the way they work.

10.1.2 Code Reading

We found no significant differences in students' ability to analyze code in a static fashion (i.e. without mental tracing). For example, our interventions did not have an effect on students' ability to determine whether a given function was tail recursive, even though Labster was able to provide explanations to students of why functions they wrote were or were not tail recursive. An intuitive explanation for the lack of impact here is that the core benefit of a visualization tool like Labster, providing a view into the notional machine that otherwise works as a black-box to students, is not crucial for understanding static properties of source code. Patterns in source code, like whether a function is tail recursive or not, are regularly accessible to students without the need for a visualization system. Consequentially there is much less room for a tool like Labster to make a difference. This intuition is supported by our data, which shows that students are able to perform very well on questions testing static analysis of source code, regardless of the use of Labster.

We did find some measures of students' ability to read code revealed significant differences. Students' ability to predict the outcome of code snippets with expressions involving arrays and pointers was found to be improved by using Labster compared to students who had completed the same activities without the tool. Completing this exercise correctly required students to mentally trace through the dynamic execution of code, which is different than several of our other code-reading exercises (e.g. "Is this function tail recursive?") for which an approach based on matching patterns in source code would be fairly successful. Intuitively, exercises that require students to actually "run" their mental models of the notional machine should be more sensitive to improvements in those models, and this trend is reflected in our results.

10.1.3 Code Writing

We found that students' performance was significantly different on only some of our tests of code writing ability. In our comparison of navigation controls, only one of the two code writing tasks given to students revealed a significant difference (and only for the subset of students with relatively high confidence). This question required students to write a recursive version of an iteratively implemented function, whereas the one that did not show a difference asked students to implement a recursive function based on given pseudocode. Arguably, performance on the question in which students had to determine the recursive structure would more truly reflect students understanding of recursive code, whereas translating an algorithm that is spelled out already does not depend heavily on students mental model of how recursion actually works at runtime.

In our comparison of the *code stack* and *static source* models of subcall execution, however, none of the code writing questions given to students directly after the lab activities showed significant differences. One of these was the same question that had shown a significant difference based on the version of navigation controls. The code students needed to write followed a pattern quite similar to other exercises students had worked through during the lab activities, and it may be that as long as students were able to complete the lab adequately, students could apply the same coding patterns to answer the question. Thus, this question would be sensitive to whether students were able to complete the lab exercises adequately, even if it is not as sensitive to how well they really understand recursion. Very basic navigation controls may well have interfered with students ability to work through the lab exercises. Our personal experience with students during the experiment suggests this was the case, as well as their less positive evaluation of the exercises and the Labster system. This was evidently not the case with different conceptual models of subcall execution, as students were able to answer similar questions equally well regardless of the model they worked with. There was a significant difference in performance on a midterm question involving recursion between groups who had worked with the *code stack* and *static source* models. This question involved a much greater degree of transfer than other code writing questions, and is discussed further in the next section.

10.1.3.1 Transfer Questions and Implications

Throughout our experiments, the general trend is that questions involving a high amount of knowledge transfer were more likely to be significantly impacted by students use of Labster. A transfer question is one that relies on the underlying conceptual knowledge students are tasked with learning, but for which the problem solving process is not essentially the same as for questions students have seen previously. Transfer questions allow us to better judge whether meaningful learning has taken place, because they cannot be answered with a fragile schema developed to understand the very narrow group of exercises students have directly worked with in the past. For our specific purposes, a transfer question involving code reading and/or writing should be difficult to answer if the student has not developed a viable mental model of the notional machine that can be generalized to reason about novel problems.

As mentioned previously, tests of students' conceptual recall or ability to analyze source code in a static fashion (i.e. without mental tracing) did not reveal statistically significant differences between our interventions. These kinds of questions involve a very low degree of transfer, because they essentially rely on the memorization of specific facts, rules, or patterns. Once these are presented to students, it is hard to design questions that probe an understanding of the notional machine but are difficult to answer just by rote memorization.

In our comparison of the *code stack* and *static source* conceptual models of subcall execution

during recursion lab exercises, we found no significant difference in students' performance on post-test code-writing questions that were similar to the linear recursive problems they had solved during the exercises. It is plausible that students' efforts to answer these questions were dominated by an application of a schema learned to write specific patterns of source code rather than a fundamental understanding of recursion. However, there was a significant difference in students' performance on a midterm question that required the use of recursion to process trees - something drastically different than the linear recursive functions students had written during the lab activities. This question involves a much higher amount of transfer, because it relies on the same foundational knowledge (i.e. how function calls and recursion work) as previously seen examples, but applied in a fundamentally different way.

10.1.4 Student Confidence and Subjective Evaluation

We found that students who used Labster during the Arrays and Pointers Lab had improved confidence with the material compared to those who had done the same exercises without the tool, and also felt more strongly that the lab exercises were effective. Additionally, we found that certain aspects of different versions of Labster produced significant differences in these same outcomes. In particular, these outcomes were drastically improved for students who had access to full navigation controls over those with only very basic controls. In our comparison of the *code stack* and *static source* models of recursion, we also found the *code stack* tended to produce more positive outcomes, but only some questions were individually statistically significant. At a high-level, the implications are that using a program visualization and the design of that visualization can improve students' confidence.

Generally, increases in objective measures of understanding (e.g. performance on code reading/writing problems) were accompanied by increases in subjective measures of confidence and more positive evaluations of the lab activities and Labster itself. The converse was not as uniformly true, but in each of our investigations at least one objective measure showed a significant difference to parallel improvements in students' subjective experience. A possible explanation for the relatively higher sensitivity of our subjective measures is that students are fairly well cognizant of their level of understanding or how much it was improved by educational activities, but it is more difficult to design objective measures that truly probe this understanding, especially if the improvements are small or incremental.

A potential objection to the validity of our subjective measures of learning is that students may simply feel like they should have learned more when engaging in an educational activity that is perceived as new or innovative. This could certainly be the case for using program visualization. Students would not necessarily even be aware of this effect, as they may legitimately feel like they

have gained greater understanding, especially because visualization ostensibly provides insights that have not been seen before, even if they are not internalized by the student. Students may feel like they finally have an insight into how code works, but when not working in Labster still have difficulties visualizing execution in their head. However, in each of our interventions, the detection of improvements in objective performance suggest the subjective improvements are based in legitimate learning. Additionally, in our experiments comparing difference conceptual models, both groups used the Labster system. Any effect due to the novelty of using a visualization system should affect both groups similarly, yet we still found significant and often large differences.

10.2 The Impact of Student Confidence

Across several of our experiments, students who came into the lab with a higher level of confidence were more sensitive to the aspect of the visualization that was varied. We identified two groups of students based on whether they reported an average confidence level of 3 or 4 (on a scale of 1 to 5) on the pre-lab survey. We often found significant results for the group of students with pre-lab confidence level 4, but not for the group with pre-lab confidence level 3.

In our investigation of navigation controls, students who entered the lab with confidence level 4 had significantly higher confidence after working with the version of Labster that had a full suite of navigation controls than those who worked with minimal controls, but the same effect was not present for students with pre-lab confidence level 3. Additionally, only those students with pre-lab confidence 4 showed a significant benefit from fuller navigation controls in their performance on post-lab code-writing tasks. In our experiment comparing different conceptual models of recursion, we also found that although the *code stack* model helped all students perform better on a difficult transfer question, the difference was more pronounced for students with higher confidence.

One potential explanation is that weaker students may just not be able to integrate the experiences they have with the visualization into their current understanding quickly enough. The knowledge construction process is one of incremental refinement, and the current state of a student's mental model of the notional machine serves as the starting point for learning. Students coming into an educational activity with a less developed mental model may simply need more time interacting with a program visualization in order to make significant progress toward a viable mental model. Especially in tests of programming ability, the difference in performance between students with viable and non-viable understanding is vast - often the code they write simply works correctly or it does not. It is harder to detect whether one non-viable understanding is in some way worse than another based only on a student's code as a product of that understanding.

On the other hand, confidence did not influence whether a significant difference was found in the way students evaluated the effectiveness of the lab activities or the Labster system itself.

Of course, we can also make no meaningful claim about whether confidence affected students' sensitivity to our interventions when there were no significant differences found at all.

10.2.1 Adding Training Wheels to Labster

Another possible explanation for the smaller benefits of using Labster observed in less confident students is that they may have been more easily overwhelmed by the visualization. If the less confident students tended to have weaker understandings of the notional machine before working with Labster, they would not as readily be able to process and incorporate information from the visualization. In particular, they may not know what to pay attention to, or may not be able to determine the optimal way to work through the execution of a program or identify particular points of interest throughout the course of the visualization. For example, students could be tripped up by clicking on the wrong navigation button, which might take them to a point of execution they didn't intend to view and require a lot of time to get back on track. A potential approach for addressing these concerns would be to add "training wheels" [119] to Labster in order to prevent beginning users or students who are less confident from running into complexities that could hinder the effectiveness of the visualization.

It would be possible to reduce the variety of navigation controls offered by Labster in an attempt to simplify the interface presented to new users. Some of the advanced navigation buttons could cause confusion if used accidentally or incorrectly, especially those that perform larger jumps in execution (e.g. "step over" or "step out"). However, evidence from our study of navigation controls suggests a careful approach is necessary. In our experiments, a very basic control scheme was clearly frustrating to users, since they evaluated the lab exercises and Labster itself much less positively than those using full controls. Further work is needed to investigate which controls were most often used by students with the full interface, and to determine which were sorely missed by those with the basic controls. A more measured approach than completely blocking these actions could be taken, either by asking for confirmation before significantly large jumps in navigation or providing an "undo" mechanism in case of accidental navigation.

Another possibility would be to only offer a larger step grain to new users of the system, but this is at odds with the principles of *completeness* and *continuity* as well as the general goal of illuminating the inner workings of the notional machine. Changing the step grain to the statement level, for example, hides many important details of how code actually works. Another, more nuanced approach would be to strategically hide individual steps for more complex operations. Seeing all the individual steps for these operations may be unproductive or even overwhelming for novice programmers who do not have a mental model of the notional machine that is well-developed enough to appropriately frame their nuances. For example, the step-by-step evaluation of each piece of a

print statement may not be important for novice programmers, yet it increases both the size and complexity of the visualization. Future studies could attempt to assess the relevance of each step in a visualization and also evaluate the impact when the least relevant are omitted.

Finally, we can also consider adding “training wheels” instead to the conceptual models and representations presented by Labster. While our individual experiments found no specific evidence that certain conceptual models were most effective for one group of students but not for another, it make intuitive sense that hiding details could be helpful in some cases. As educators, we often provide students with simplified representations in order to provide intuition, streamline learning, or temporarily hide away tricky details. Full details can be provided once students have become accustomed to the concepts at hand and have built a foundation on which more complex understanding can be constructed. For example, some of these considerations are already built into the way Labster displays the contents of memory. By default, some objects in memory (e.g. string literals, temporary objects, globals not created by the user) are hidden from students because they clutter up the display and get in the way of what students actually want to focus on. Additional options could also be provided for advanced users to access a display of this memory, but the key is to present a manageable view for students. A future investigation comparing Labster’s current approach with a more complete view of memory would be a good first step toward quantifying the impact of these training wheels.

10.3 Code-Context Visualization in Labster

One of the major design decisions in Labster’s visualization of the notional machine is to use a dynamic representation of the code itself as concrete and authentic medium for visualization and interaction. We call this approach *code-context visualization*. It can be contrasted against the use of visual metaphors or other abstract representations in many contemporary visualization systems. While these can be used to convey the conceptual models behind programming concepts to students, students attempting to understand and write real programs will be reading and writing actual code rather than working with visual metaphors or abstract representations.

The approach of code-context visualization is wherever possible to “bring the code to life” to illustrate execution. This naturally applies to our model of “in situ” expression evaluation. Instead of showing expression evaluation using abstract graphics or in a separate evaluation area, we simply rewrite the terms of an expression directly in-place as it executes piece by piece. The *code stack* model of subcall execution also follows the principle, because we simply display another instance of the code for each function call that is made and stack them up in parallel to the set of stack frames in memory.

These two specific examples of code-context visualization complement each other well. If

students are working with a program that has made several function calls, the context to which each call in the code stack will return is made evident by the presence of expressions that have been partially evaluated in-situ. For example, in a recursive call chain, students are easily able to discern which computations have been or will be performed by each different call in the recursive call chain. The visualization gives them all the information needed to reason about the way each individual invocation of the function contributes to an overall recursive computation. The results of our experimental investigations, which show that both the *code stack* model and “in situ” expression evaluation can positively impact learning, provide initial pieces of support for *code-context visualization* as a general design principle in program visualization.

CHAPTER 11

Conclusion

In this dissertation, we have investigated several design choices in program visualization systems with the goal of illuminating the notional machine so that students may learn from richer, more meaningful experiences running the programs they write.

To achieve this, we created the Labster program visualization system, which has served both as an educational tool and a platform for our experimental investigations into program visualization techniques. The Labster system brings together successful approaches previously described in the program visualization literature, but also includes several novel techniques like “in situ” expression evaluation, the *code stack* model of subcall execution, “point+click” navigation controls, and static/dynamic analysis for improved feedback about the behavior of students’ code.

We integrated interactive exercises using Labster into the coursework for EECS 280, and studied students experiences with the system and its impact on their learning. Overall, we found that using the tool has a significant impact on students’ learning outcomes. In particular, students who used Labster showed improvements in confidence and tests of code-tracing proficiency over those who completed the same learning activities without visualization.

We also found that the navigation controls available to students can have a profound impact on their learning experiences. In a comparison between very basic single-step forward/backward navigation and a full fledged set of controls, we found the full featured version to be superior. Students who worked with full navigation controls showed improved performance on tests of code-writing proficiency, were more confident in their understanding, viewed the lab exercises as more effective, and gave more positive reviews of the Labster system.

Students who worked with Labster’s *code stack* model of subcall execution evaluated the system more positively and had improved performance on an exam question concerning recursion compared to students who had worked with the *static source* model used in many contemporary visualization systems. This, as well as the effectiveness of “in-situ” expression evaluation for

improving students code-tracing abilities suggest the general design principle of *code-context visualization* and its use in Labster are promising.

Finally, the educational impact we were able to detect in our study is made more impressive by the fact that interaction with Labster was only a very small part of the instruction given to students in the course. For example, students' ability to write recursive code on the midterm exam drew on their experiences in lecture, working on projects, asking questions in office hours, and studying on their own. Despite all these other influences, there was still a significant difference in performance depending on the version of Labster students used.

BIBLIOGRAPHY

- [1] A. E. Tew and M. Guzdial, “The fcs1: A language independent assessment of cs1 knowledge,” in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’11. New York, NY, USA: ACM, 2011, pp. 111–116. [Online]. Available: <http://doi.acm.org/10.1145/1953163.1953200>
- [2] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming,” *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007.
- [3] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä *et al.*, “A multi-national study of reading and tracing skills in novice programmers,” *ACM SIGCSE Bulletin*, vol. 36, no. 4, pp. 119–150, 2004.
- [4] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, “A multi-national, multi-institutional study of assessment of programming skills of first-year cs students,” *ACM SIGCSE Bulletin*, vol. 33, no. 4, pp. 125–180, 2001.
- [5] E. Soloway, “Learning to program= learning to construct mechanisms and explanations,” *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.
- [6] A. Venables, G. Tan, and R. Lister, “A closer look at tracing, explaining and code writing skills in the novice programmer,” in *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ser. ICER ’09. New York, NY, USA: ACM, 2009, pp. 117–128. [Online]. Available: <http://doi.acm.org/10.1145/1584322.1584336>
- [7] M. Lopez, J. Whalley, P. Robbins, and R. Lister, “Relationships between reading, tracing and writing skills in introductory programming,” in *Proceedings of the Fourth international Workshop on Computing Education Research*. ACM, 2008, pp. 101–112.
- [8] J. C. Spohrer and E. Soloway, “Novice mistakes: Are the folk wisdoms correct?” *Communications of the ACM*, vol. 29, no. 7, pp. 624–632, 1986.
- [9] A. Robins, J. Rountree, and N. Rountree, “Learning and teaching programming: A review and discussion,” *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.
- [10] A. Gomes and A. J. Mendes, “Learning to program-difficulties and solutions,” in *International Conference on Engineering Education-ICEE*, vol. 2007, 2007.

- [11] D. B. Palumbo, “Programming language/problem-solving research: A review of relevant issues,” *Review of Educational Research*, vol. 60, no. 1, pp. 65–89, 1990.
- [12] B. Du Boulay, T. O’Shea, and J. Monk, “The black box inside the glass box: presenting computing concepts to novices,” *International Journal of Man-Machine Studies*, vol. 14, no. 3, pp. 237–249, 1981.
- [13] M. Ben-Ari, “Constructivism in computer science education,” *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.
- [14] J. Sorva, “Notional machines and introductory programming education,” *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 2, p. 8, 2013.
- [15] B. D. Boulay, “Some difficulties of learning to program,” *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986.
- [16] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide, “Exploring the role of visualization and engagement in computer science education,” in *ACM SIGCSE Bulletin*, vol. 35, no. 2. ACM, 2002, pp. 131–152.
- [17] B. A. Price, R. M. Baecker, and I. S. Small, “A principled taxonomy of software visualization,” *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, 1993.
- [18] J. Sorva, V. Karavirta, and L. Malmi, “A review of generic program visualization systems for introductory programming education,” *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, p. 15, 2013.
- [19] T. Lauer, “Reevaluating and refining the engagement taxonomy,” in *ACM SIGCSE Bulletin*, vol. 40, no. 3. ACM, 2008, pp. 355–355.
- [20] N. Myller, R. Bednarik, E. Sutinen, and M. Ben-Ari, “Extending the engagement taxonomy: Software visualization and collaborative learning,” *ACM Transactions on Computing Education (TOCE)*, vol. 9, no. 1, p. 7, 2009.
- [21] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, “A survey of successful evaluations of program visualization and algorithm animation systems,” *ACM Transactions on Computing Education (TOCE)*, vol. 9, no. 2, p. 9, 2009.
- [22] P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander, “Taxonomy of effortless creation of algorithm visualizations,” in *Proceedings of the first international workshop on Computing education research*. ACM, 2005, pp. 123–133.
- [23] B. S. Bloom, M. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl, “Taxonomy of educational objectives: Handbook i: Cognitive domain,” *New York: David McKay*, vol. 19, p. 56, 1956.
- [24] L. W. Anderson, D. R. Krathwohl, and B. S. Bloom, *A taxonomy for learning, teaching, and assessing: A revision of Bloom’s taxonomy of educational objectives*. Allyn & Bacon, 2001.

- [25] E. Thompson, A. Luxton-Reilly, J. L. Whalley, M. Hu, and P. Robbins, “Bloom’s taxonomy for cs assessment,” in *Proceedings of the tenth conference on Australasian computing education-Volume 78*. Australian Computer Society, Inc., 2008, pp. 155–161.
- [26] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. Kumar, and C. Prasad, “An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies,” in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 2006, pp. 243–252.
- [27] R. Lister and J. Leaney, “Introductory programming, criterion-referencing, and bloom,” in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 143–147.
- [28] C. G. Johnson and U. Fuller, “Is bloom’s taxonomy appropriate for computer science?” in *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. ACM, 2006, pp. 120–123.
- [29] U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, J. Jackova, E. Lahtinen, T. L. Lewis, D. M. Thompson, C. Riedesel *et al.*, “Developing a computer science-specific learning taxonomy,” *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 152–170, 2007.
- [30] M. Prince, “Does active learning work? a review of the research,” *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.
- [31] R. R. Hake, “Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses,” *American journal of Physics*, vol. 66, p. 64, 1998.
- [32] J. Sajaniemi, M. Kuittinen, and T. Tikansalo, “A study of the development of students’ visualizations of program state during an elementary object-oriented programming course,” *Journal on Educational Resources in Computing (JERIC)*, vol. 7, no. 4, p. 3, 2008.
- [33] S. Papert, *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.
- [34] M. J. Prince and R. M. Felder, “Inductive teaching and learning methods: Definitions, comparisons, and research bases,” *Journal of engineering education*, vol. 95, no. 2, pp. 123–138, 2006.
- [35] M. A. Albanese and S. Mitchell, “Problem-based learning: A review of literature on its outcomes and implementation issues,” *Academic medicine*, vol. 68, no. 1, pp. 52–81, 1993.
- [36] J. M. Keller, “Strategies for stimulating the motivation to learn,” *Performance+ Instruction*, vol. 26, no. 8, pp. 1–7, 1987.
- [37] J. Sweller, “Cognitive load during problem solving: Effects on learning,” *Cognitive science*, vol. 12, no. 2, pp. 257–285, 1988.
- [38] J. Sweller, J. J. Van Merriënboer, and F. G. Paas, “Cognitive architecture and instructional design,” *Educational psychology review*, vol. 10, no. 3, pp. 251–296, 1998.

- [39] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information." *Psychological review*, vol. 63, no. 2, p. 81, 1956.
- [40] A. J. Ko and B. A. Myers, "Development and evaluation of a model of programming errors," in *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*. IEEE, 2003, pp. 7–14.
- [41] D. Shaffer, "Cohesion, coupling, and abstraction," *Encyclopedia of Information Systems*, pp. 127–139, 2003.
- [42] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, "Knowledge organization and skill differences in computer programmers," *Cognitive Psychology*, vol. 13, no. 3, pp. 307–325, 1981.
- [43] N. J. Cooke and R. W. Schvaneveldt, "Effects of computer programming experience on network representations of abstract programming concepts," *International Journal of Man-Machine Studies*, vol. 29, no. 4, pp. 407–427, 1988.
- [44] B. Adelson, "Problem solving and the development of abstract categories in programming languages," *Memory & cognition*, vol. 9, no. 4, pp. 422–433, 1981.
- [45] R. Brooks, "Towards a theory of the comprehension of computer programs," *International journal of man-machine studies*, vol. 18, no. 6, pp. 543–554, 1983.
- [46] R. E. Mayer, "Multimedia learning," *Psychology of Learning and Motivation*, vol. 41, pp. 85–139, 2002.
- [47] R. E. Mayer and R. Moreno, "Nine ways to reduce cognitive load in multimedia learning," *Educational psychologist*, vol. 38, no. 1, pp. 43–52, 2003.
- [48] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.
- [49] R. E. Mayer, M. Hegarty, S. Mayer, and J. Campbell, "When static media promote active learning: annotated illustrations versus narrated animations in multimedia instruction." *Journal of Experimental Psychology: Applied*, vol. 11, no. 4, p. 256, 2005.
- [50] R. C. Clark and R. E. Mayer, *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. Wiley. com, 2011.
- [51] D. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, E. Soloway and S. Iyengar, Eds. Ablex Publishing Corp., 1986, pp. 213–229.
- [52] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008.

- [53] T. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J. R. Ruthruff, L. Beckwith, and A. Phalgune, “Impact of interruption style on end-user debugging,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 287–294.
- [54] C. S. Peirce, “(1998). on the logic of drawing history from ancient documents, especially from testimonies,” 1998). *The essential Peirce: Selected philosophical writings*, vol. 2, pp. 75–114, 1901.
- [55] R. L. Mack, C. H. Lewis, and J. M. Carroll, “Learning to use word processors: problems and prospects,” *ACM Transactions on Information Systems (TOIS)*, vol. 1, no. 3, pp. 254–271, 1983.
- [56] J. Taylor, “Analysing novices analysing prolog: what stories do novices tell themselves about prolog?” *Instructional Science*, vol. 19, no. 4-5, pp. 283–309, 1990.
- [57] R. D. Pea, “Language-independent conceptual” bugs” in novice programming,” *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 25–36, 1986.
- [58] M. J. Oudshoorn, H. Widjaja, and S. Ellershaw, “Aspects and taxonomy of program visualisation,” *Software Visualisation*, vol. 7, pp. 3–26, 1996.
- [59] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards, “Algorithm visualization: The state of the field,” *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 3, p. 9, 2010.
- [60] J. B. Bennedsen and C. Schulte, “Bluej visual debugger for learning the execution of object-oriented programs?” *ACM Transactions on Computing Education*, vol. 10, no. 2, 2010.
- [61] T. Lauer, “Learner interaction with algorithm visualizations: viewing vs. changing vs. constructing,” in *ACM SIGCSE Bulletin*, vol. 38, no. 3. ACM, 2006, pp. 202–206.
- [62] J. Urquiza-Fuentes and J. A. Velázquez-Iturbide, “An evaluation of the effortless approach to build algorithm animations with winhipe,” *Electron. Notes Theor. Comput. Sci.*, vol. 178, pp. 3–13, Jul. 2007. [Online]. Available: <http://dx.doi.org/10.1016/j.entcs.2007.01.038>
- [63] S. Grissom, M. F. McNally, and T. Naps, “Algorithm visualization in cs education: comparing levels of student engagement,” in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 87–94.
- [64] S. R. Hansen, N. H. Narayanan, and D. Schrimpsheer, “Helping learners visualize and comprehend algorithms,” *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, vol. 2, no. 1, p. 10, 2000.
- [65] C. Hundhausen and S. Douglas, “Using visualizations to learn algorithms: should students construct their own, or view an expert’s?” in *Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on*. IEEE, 2000, pp. 21–28.
- [66] C. D. Hundhausen and J. L. Brown, “Designing, visualizing, and discussing algorithms within a cs 1 studio experience: An empirical study,” *Computers & Education*, vol. 50, no. 1, pp. 301–326, 2008.

- [67] T. Hübscher-Younger and N. H. Narayanan, “Dancing hamsters and marble statues: characterizing student visualizations of algorithms,” in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 95–104.
- [68] B. A. Price, *A framework for the automatic animation of concurrent programs*, Unpublished M.S. thesis. Department of Computer Science, University of Toronto., 1991.
- [69] A. W. Lawrence, “Empirical studies of the value of algorithm animation in algorithm understanding,” 1993.
- [70] J. S. Gurka, *Pedagogic aspects of algorithm animation*. University of Colorado at Boulder, 1996.
- [71] P. Mulholland, “A principled approach to the evaluation of sv: a case study in prolog,” *Software visualization: Programming as a multimedia experience*, pp. 439–452, 1998.
- [72] C. Pareja-Flores, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide, “Winhipe: an ide for functional programming based on rewriting and visualization,” *ACM SIGPLAN Notices*, vol. 42, no. 3, pp. 14–23, 2007.
- [73] V. Karavirta and C. A. Shaffer, “Jsav: the javascript algorithm visualization library,” in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 2013, pp. 159–164.
- [74] C. A. Shaffer, V. Karavirta, A. Korhonen, and T. L. Naps, “Opensa: beginning a community active-ebook project,” in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. ACM, 2011, pp. 112–117.
- [75] T. Sirkiä, “A javascript library for visualizing program execution,” in *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. ACM, 2013, pp. 189–190.
- [76] P. Smith and G. Webb, “Reinforcing a generic computer model for novice programmers,” in *Proceedings of the Seventh Australian Society for Computers in Learning in Tertiary Education Conference (ASCILITE’95)*. ASCILITE.
- [77] P. A. Smith and G. I. Webb, “Transparency debugging with explanations for novice programmers.” in *AADEBUG*. Citeseer, 1995, pp. 105–118.
- [78] M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen, “A decade of research and development on program animation: The jeliot experience,” *Journal of Visual Languages & Computing*, vol. 22, no. 5, pp. 375–384, 2011.
- [79] P. J. Guo, “Online python tutor: embeddable web-based program visualization for cs education,” in *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 2013, pp. 579–584.
- [80] C. E. George, “Investigating the effectiveness of a software-reinforced approach to understanding recursion.” Ph.D. dissertation, Goldsmiths College (University of London), 1996.

- [81] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen, "The jeliot 2000 program animation system," *Computers & Education*, vol. 40, no. 1, pp. 1–15, 2003.
- [82] J. A. Velázquez-Iturbide, "Principled design of logical fisheye views of functional expressions," *SIGPLAN Not.*, vol. 41, no. 8, pp. 34–43, Aug. 2006. [Online]. Available: <http://doi.acm.org.proxy.lib.umich.edu/10.1145/1163566.1163575>
- [83] M. Thuné and A. Eckerdal, "Variation theory applied to students conceptions of computer programming," *European Journal of Engineering Education*, vol. 34, no. 4, pp. 339–347, 2009.
- [84] J. Sorva, "Reflections on threshold concepts in computer programming and beyond," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 2010, pp. 21–30.
- [85] J. H. Meyer and R. Land, "Threshold concepts and troublesome knowledge," *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge*, pp. 3–18, 2006.
- [86] D. A. Norman, "Some observations on mental models," *Mental models*, vol. 7, no. 112, pp. 7–14, 1983.
- [87] R. E. Mayer, "The psychology of how novices learn computer programming," *ACM Computing Surveys (CSUR)*, vol. 13, no. 1, pp. 121–141, 1981.
- [88] R. M. Young, "The machine inside the machine: users' models of pocket calculators," *International Journal of Man-Machine Studies*, vol. 15, no. 1, pp. 51–85, 1981.
- [89] —, "Surrogates and mappings: Two kinds of conceptual models for interactive devices," *Mental models*, vol. 37, pp. 35–52, 1983.
- [90] J. J. Cañas, M. T. Bajo, and P. Gonzalvo, "Mental models and computer programming," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 795–811, 1994.
- [91] R. E. Mayer, "Different problem-solving competencies established in learning computer programming with and without meaningful models." *Journal of Educational Psychology*, vol. 67, no. 6, p. 725, 1975.
- [92] —, "Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order." *Journal of Educational Psychology*, vol. 68, no. 2, p. 143, 1976.
- [93] C. E. George, "Using visualization to aid program construction tasks," in *ACM SIGCSE Bulletin*, vol. 34, no. 1. ACM, 2002, pp. 191–195.
- [94] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, "Ville: a language-independent program visualization tool," in *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*. Australian Computer Society, Inc., 2007, pp. 151–159.

- [95] M. P. Bruce-Lockhart and T. S. Norvell, "Developing mental models of computer programming interactively via the web," in *Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE'07. 37th Annual. IEEE*, 2007, pp. S3H-3.
- [96] J. Sundararaman and G. Back, "Hdpv: interactive, faithful, in-vivo runtime state visualization for c/c++ and java," in *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 2008, pp. 47-56.
- [97] "Codemirror." [Online]. Available: <http://codemirror.net>
- [98] M. J. Lee and A. J. Ko, "Personifying programming tool feedback improves novice programmers' learning," in *Proceedings of the seventh international workshop on Computing education research*. ACM, 2011, pp. 109-116.
- [99] I. Milne and G. Rowe, "Difficulties in learning and teaching programming views of students and tutors," *Education and Information technologies*, vol. 7, no. 1, pp. 55-66, 2002.
- [100] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," in *ACM SIGCSE Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 14-18.
- [101] D. Boud and N. Falchikov, "Quantitative studies of student self-assessment in higher education: A critical analysis of findings," *Higher education*, vol. 18, no. 5, pp. 529-549, 1989.
- [102] R. E. Clark, "Antagonism between achievement and enjoyment in ati studies," *Educational Psychologist*, vol. 17, no. 2, pp. 92-101, 1982.
- [103] P. R. Ventura, "Identifying predictors of success for an objects-first cs1," *Computer Science Education*, vol. 15, pp. 223-243, 2005.
- [104] R. Boyle, J. Carter, and M. Clark, "What makes them succeed? entry, progression and graduation in computer science," *Journal of Further and Higher Education*, vol. 26, no. 1, pp. 3-18, 2002.
- [105] B. C. Wilson, "A study of factors promoting success in computer science including gender differences," *Computer Science Education*, vol. 12, no. 1-2, pp. 141-164, 2002.
- [106] G. Scragg and J. Smith, "A study of barriers to women in undergraduate computer science." in *ACM SIGCSE Bulletin*, vol. 30, no. 1. ACM, 1998, pp. 82-86.
- [107] T. Sirkiä, "Exploring expression-level program visualization in cs1," in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. ACM, 2014, pp. 153-157.
- [108] H. Lieberman and C. Fry, "Zstep 95: A reversible, animated source code stepper," *Software Visualization: Programming as a Multimedia Experience*, pp. 277-292, 1997.

- [109] L. M. Mann, M. C. Linn, and M. Clancy, “Can tracing tools contribute to programming proficiency? the lisp evaluation modeler,” *Interactive Learning Environments*, vol. 4, no. 1, pp. 096–113, 1994.
- [110] G. Weber and P. Brusilovsky, “Elm-art: An adaptive versatile system for web-based instruction,” *International Journal of Artificial Intelligence in Education (IJAIED)*, vol. 12, pp. 351–384, 2001.
- [111] M. Auguston and J. Reinfelds, “A visual miranda machine,” in *Software Education Conference, 1994. Proceedings.* IEEE, 1994, pp. 198–203.
- [112] H. Kahney, “What do novice programmers know about recursion,” in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems.* ACM, 1983, pp. 235–239.
- [113] T. Götschi, I. Sanders, and V. Galpin, *Mental models of recursion.* ACM, 2003, vol. 35, no. 1.
- [114] I. Sanders, V. Galpin, and T. Götschi, “Mental models of recursion revisited,” in *ACM SIGCSE Bulletin*, vol. 38, no. 3. ACM, 2006, pp. 138–142.
- [115] I. D. Sanders and V. C. Galpin, “Students’ mental models of recursion at wits,” in *ACM SIGCSE Bulletin*, vol. 39, no. 3. ACM, 2007, pp. 317–317.
- [116] T. L. Scholtz and I. Sanders, “Mental models of recursion: investigating students’ understanding of recursion,” in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education.* ACM, 2010, pp. 103–107.
- [117] C. E. George, “Erosivisualising recursion and discovering new errors,” in *ACM SIGCSE Bulletin*, vol. 32, no. 1. ACM, 2000, pp. 305–309.
- [118] J. Engblom, “A review of reverse debugging,” in *System, Software, SoC and Silicon Debug Conference (S4D), 2012.* IEEE, 2012, pp. 1–6.
- [119] J. M. Carroll and C. Carrithers, “Training wheels in a user interface,” *Communications of the ACM*, vol. 27, no. 8, pp. 800–806, 1984.