Step-by-Step or Skip a Few
A Review of Program Visualizers and Recommendations for Future Work

Jacob Valero
Advisor: Dave Wonnacott

Fall 2021
Contents

1 Abstract 2

2 Introduction 2

3 The Problem 3

4 Notional Machines 4

5 Program Visualizers 5

6 Properties of Good Program Visualizers 6
   6.1 Jeliot 6
   6.2 Python Tutor 8

7 Improving Program Visualizers 10
   7.1 Balancing Accessibility and Customizability 10
   7.2 Engagement 10

8 Proposed Work 12
   8.1 The Abstract Program Visualizer 12
   8.2 Important Characteristics 13

9 Conclusion 14

10 References 14
1 Abstract

Introductory computer science students have trouble building consistent mental models of computer execution. To alleviate this problem, previous work suggests explicitly introducing a consistent and accurate abstraction of program execution, commonly referred to as a notional machine, in introductory computer science courses. One popular method for helping students understand notional machines is through program visualizers, software used to visualize code execution. This thesis will review the current state of program visualizers, the properties of widely used and effective program visualizers, and the limitations of program visualizers at the moment. Finally, suggestions for future work to improve upon current limitations will be proposed.

2 Introduction

Learning how to program is difficult, as found by numerous studies as far back as the 1980s and until more recently [MFRW07, Sor12]. In order to tackle this issue, many papers have suggested teaching the concept of a notional machine [Bou86, DBB20]. A notional machine is a consistent and accurate abstraction of program execution [Bou86, DBB20]. Papers which advocate for its use suggest explicitly introducing the concept of a notional machine in beginner computer science (CS) courses so that students’ mental models of program execution are accurately formed from the start [Bou86, DBB20].

A popular method for teaching a notional machine is through visualizing program execution, which can be done through program visualizers [Sor12]. Program visualizers are software which abstractly display what happens in the memory of the computer when a program is executed [Sor12].

This thesis aims to summarize the current state of program visualizers, evaluate what makes them effective, and analyze current limitations of current program visualizers in order to propose future work. In particular, creating a program visualizer which allows for customizing which parts of a
program will be visualized could be useful for visualizing recursion, a notoriously difficult concept for introductory CS students

3 The Problem

Numerous studies have concluded that many students leave introductory computer science courses (CS1) unable to program due to misunderstanding core computer science fundamentals [Sor12]. For example, one study conducted across multiple countries and institutions found that on an exam assessing the ability of first year computer science students, the average score was 22.89 out of 110 points [MFRW07]. This is one of many studies which reach similar conclusions, as such, many have investigated what may be causing these difficulties in learning how to program [Bou86, Sor12, Sor13].

One line of inquiry suggests that students lack consistent and accurate mental models of how code executes [MFRW07, Bou86]. One study assessing students’ models of value and reference assignment found that only 17% of the students held a viable mental model for these topics [MFRW07]. Furthermore, they found that the students who did hold a viable mental model performed significantly better in the course compared to students with non-viable mental models [MFRW07]. Understanding of value and reference assignment are just a few of many misconceptions commonly held by CS1 students [Sor12].

The idea of inaccurate mental models being a barrier to CS1 educational outcomes goes as far back as 1986 [Bou86]. DuBoulay suggested that these inaccurate mental models formed as students were not provided with a correct model of computer execution at the start of their CS1 courses [Bou86]. As a proposed solution, DuBoulay introduced the idea of an abstract model that instructors could introduce to CS1 students, which he termed the notional machine [Bou86].
Notional Machines

The notional machine was first defined by DuBoulay as the general properties of the machine that programmers try to control [Bou86]. The purpose of the notional machine is to define and better understand what will happen when a program executes [Sor13]. Sorva also most recently defined it as an abstract computer which executes code [Sor13]. A notional machine is an abstraction as it hides the underlying details of how computers actually execute code, which is useful as it contains just enough detail to understand how code will execute, while hiding away information that is mostly unnecessary when writing code [Sor13]. It is important to note that there is not just one notional machine, as there are many different ways a computer can be abstracted [Sor12]. For example, notional machines can be defined for different languages or programming paradigms [Sor12]. To further clarify the definition, it is also good to understand what notional machines are not. Notional machines are not mental models of computers, nor are they visualizations of the computer, although the two concepts are closely related [Sor13]. The mental models programmers have of program execution can be thought of the mental models of notional machines, while visualizations can be useful in demonstrating how a notional machine works, as will be discussed in this paper [Sor13].

As stated earlier, a major stumbling block for many students is the inaccurate mental models they hold of notional machines. These inaccurate models develop as a result of students reaching their own conclusions as to how a notional machine works, as seen in the many misconceptions held by students [Bou86, Sor12]. Although CS1 courses implicitly refer to some abstraction of the computer when teaching programming, rarely do they introduce a notional machine explicitly, which may lead to the common misconceptions and inaccurate mental models held by students [Sor13]. As such, explicitly teaching about a notional machine in CS1 courses may help students form consistent and accurate mental models of program execution [Sor12].
5 Program Visualizers

One of the most common ways to explicitly introduce a notional machine is through program visualization [Sor13]. Program visualization is any means of visualizing the execution of code, ranging from pencil-and-paper demonstrations to using software to automatically visualize the current state of a program [Sor13].

This paper will focus on software which visualizes code execution, as they have become popular pedagogical tools with researched benefits on improving learning outcomes [Guo21, Sor13, LBAU03]. The software, known as program visualizers, present what occurs, at an abstract level, in the memory of the computer as the code executes [Sor12]. Program visualizers are particularly good at showcasing the inner-workings of the notional machine as they can visualize what is not immediately evident for novice programmers from just looking at code, such as assignment and control flow [LBAU03]. By working through examples and ideally with tailored exercises, students can familiarize themselves with the rules of a notional machine [POJ+20].

One study which supports the benefits of program visualizers was conducted with a popular program visualizer known as Jeliot [LBAU03]. In the experiment, students were split into two groups in a year-long high-school CS1 course [LBAU03]. The first group had an additional hour of content using Jeliot while the second group covered the same content but without the use of Jeliot [LBAU03]. To assess the students’ comprehension, both exams and interviews were completed [LBAU03]. The experimenters found that the group of students who used Jeliot throughout the course better understood fundamental concepts, such as assignment and control-flow, and were better at describing how programs would execute [LBAU03].

This study is one of many providing evidence of the benefits of program visualizers [Sor12]. Jeliot in particular has been in development for more than twenty years, which has allowed many studies supporting its benefits to be conducted [Sor12] Some of the other benefits found include improving
attention and increased collaboration among students \cite{babl11, lbau03, sor12}.

Program visualizers seem to support learning outcomes by visualizing a notional machine. However, these benefits are not a result of the just the visualizations by themselves \cite{babl11}. Specific design considerations go into creating program visualizers which effectively support learning outcomes \cite{babl11, guo21}. As such, it is important to cover the design principles used by program visualizers which are widely used and well-studied.

\section{Properties of Good Program Visualizers}

Two program visualizers serve as good case studies for examining the properties of program visualizers. These two visualizers are Jeliot, a visualizer for Java programs created by Mordechai Ben-Ari and others, and Python Tutor, a visualizer for Python programs created by Phillip Guo \cite{babl11, guo21}. They make particularly good examples of program visualizers as they have both been in development for long periods of time, are widely used, and have been extensively studied when compared to many other program visualizers \cite{babl11, guo21}. This section will review both visualizers and summarize why they have been so successful.

\subsection{Jeliot}

Jeliot has been continuously developed since 1997, one of the program visualizers longest in development \cite{babl11}. Jeliot was the first visualizer to introduce automatic animations, making it more accessible to many as less effort is required on the end of the user in order to create visualizations of code \cite{babl11}. Furthermore, its educational benefits, such as improving attention in the classroom and improving comprehension of key CS1 concepts, have been extensively studied \cite{babl11, lbau03, sor12}. The
creators of Jeliot have also improved upon the program throughout the years by studying what makes it effective at teaching CS1 concepts [BABL+11].

As stated earlier, Jeliot was the first program visualizer to introduce automatic animations, arguably leading to its widespread adoption [BABL+11]. All that is required in order to visualize programs is a Java program, meaning minimal effort is required to create visualizations [BABL+11]. This trait is shared by both Jeliot and Python Tutor and is cited by both development teams as being one of the key factors in having programmers use their software [BABL+11, Guo21].

Another principle important to Jeliot’s design is that as much of the data structures and control flow are present so that the inner-workings of a notional machine can properly be explained [BABL+11]. The creators state that this can be difficult to balance because it can clutter the screen, but they also found that instructors could not properly explain certain concepts without certain levels of detail [BABL+11]. For example, they originally chose to display strings simply as string values instead of objects, but then instructors could not show the difference between equating strings with ‘=’ and equals, so they added an option to display strings as objects [BABL+11]. This principle is particularly interesting because notional machines must also strike a balance as to what level of detail they should be defined [Sor13].

Jeliot also attributes factors external to the software itself as important for its success, the first being its integration with online learning environments [BABL+11]. Jeliot is able to integrate with the online learning management platform Moodle, which can be a convenient way for instructors and students to use the tool [BABL+11]. For example, instructors can create activities with discussion questions attached which can help students engage critically with the visualizations [BABL+11]. It also further increases the ease of using the software by having it accessible online, something that Python Tutor also cites as important to its success [BABL+11, Guo21]. Another external factor is the community of programmers who use and contribute to
Jeliot, as they provide feedback to improve the software [BABL^+11]. Furthermore, the authors cite that frictionless experiences with installation, technical help, and training can help immensely with adopting the use of the software [BABL^+11]. The authors go so far as to say that these features should be emphasized just as much as the design features of the program visualizer [BABL^+11]. Finally, the Jeliot team recommends that anyone creating educational software should extensively research and evaluate their tool in order to make improvements [BABL^+11].

6.2 Python Tutor

Python Tutor began development in 2009 and has now been used by over ten million people from around the world [Guo21]. The creator, Phillip Guo, has written about the design principles of program visualizers, and how to create software used in research that is sustainable and widely-used [Guo21, POJ^+20]. Sustainability is important for software used in research such as program visualizers as it allows for longer term research to be conducted, leading to a better understanding of program visualizer benefits and limitations [Guo21].

There are a number design principles that Guo put into Python Tutor, only a few of which will be covered here. The first principle is making the tool as simple as possible for a user to start using the software [Guo21]. Guo states that a user should be able to simply ”walk-up” and use the software, which ideally means users should not have to log-in to an account or install software [Guo21]. The software should prioritize robustness over the exploration of novel ideas so that users can expect a reliable tool [Guo21]. A visualizer should also minimize the amount of options available in order to reduce the amount of features a likely small team will have to maintain and to make the tool easier to use [Guo21]. Finally, on a technical level, the software should be stateless, use slower developing technologies, and minimize dependencies, in order to improve robustness and maintainability [Guo21].
In Pollock and Guo et al., the authors develop a three-axis model for the critical components program visualizers should have to effectively explain a notional machine [POJ\textsuperscript{+20}]. The paper recommends that designers of program visualizers should consider these three questions throughout the development cycle:

1. ”What is the machine’s configuration at each execution step?”
2. ”Why did an execution step take place?”
3. ”How did an execution step change the machine’s configuration?”

The authors used this model to explain instructors’ additions to Python Tutor’s visualizations [POJ\textsuperscript{+20}].

![Visualizing How Functions Work](https://python.tutor.com/visualize.html)

Figure 1: A picture from [POJ\textsuperscript{+20}] showing an instructor addition to a python tutor visualization.

For example, in Figure 1, the instructor added a comment in the lower right-hand corner to answer the second question of the three-axis model, as it may not be clear as to why the execution step took place [POJ\textsuperscript{+20}].
7 Improving Program Visualizers

It is also important to look at what is currently missing from program visualizers in order to improve them.

7.1 Balancing Accessibility and Customizability

Jeliot’s developers have numerous recommendations for future work on Jeliot and other program visualizers. Jeliot has decided to make strategic trade-offs in order to create a program visualizer which is easier to pick-up and use [BABL+11]. However, the authors cite that adding some customizability would be worth investigating [BABL+11].

For example, one critique of automatic animations is the inability to specify which lines of code should be animated [BABL+11]. This could help visualize only the most relevant portions of a program as defined by the user [BABL+11]. Similarly, an ability to control the level of visualization detail could be used to toggle between learning a more complicated notional machine or a more abstract one [BABL+11]. The creators emphasize that although these would be useful features, it is important to balance accessibility and customizability, so adding these features should be done carefully [BABL+11].

7.2 Engagement

One recommendation found in much of the literature but without widespread implementation in program visualizers is increasing engagement [BABL+11, Guo13, LBAU03, Sor12, Sor13]. It may seem that many of the benefits from program visualizers simply comes from their visualizations, but the literature is in consensus on the importance of engagement in visualizations [BABL+11, Guo13, LBAU03, Sor12, Sor13]. Sorva’s doctoral dissertation’s main thesis is on increasing engagement in program visualizers [Sor12]. He cites that increasing engagement seems promising based on existing pedagogical theory
and evidence, but testing of this idea is in order [Sor12]. Sorva has taken steps in testing this idea by creating a program visualizers called UUhistle [Sor12]. UUhistle has the option for students to take a more active role in evaluating a program by “acting” as the notional machine and executing the program themselves [Sor12]. Preliminary results from a study conducted by Sorva suggest that UUhistle had short-term educational benefits and students enjoyed using the tool, but more research is needed to better understand the importance of engagement [Sor12].

The authors of Jeliot and Python Tutor reach similar conclusions BenAri2011ADO, Guo:PythonTutor. Jeliot’s creators have cited that improving engagement is something that they hope to add to Jeliot in the future, as they have already found that the benefits of using program visualizers improve when students are more engaged with the tool [BABL+11]. Specifically, they found that the benefits of using Jeliot were most apparent when teachers prepared lessons that use Jeliot in advance [BABL+11]. One idea the authors had is to have the ability for instructors to add questions that appear during the visualization [BABL+11]. Outside of the visualizations, the authors are also experimenting with adding Jeliot integration with an online textbook so that the context of the material of the book can be supported through Jeliot’s visualizations [BABL+11].

Python Tutor’s creator Phillip Guo has stated similar recommendations [Guo13]. He recommended adding questions, annotations, quizzes, or discussions directly into the visualizations as one way to increase engagement [Guo13]. Python Tutor also has the capability of integrating with online learning materials, which supports increasing engagement outside of the tool [Guo13].
8 Proposed Work

8.1 The Abstract Program Visualizer

Another current limitation of program visualizers at the moment is that it seems that there may not be a program visualizer created specifically with teaching recursion in mind. If this is the case, then this is a particularly important gap to fill as recursion is an important CS1 concept that is notoriously hard for CS1 students to grasp [TWFK18]. There are visualizers which are created based off of a functional programming paradigm rather than an imperative one, which may have some benefits for learning recursion [TWFK18]. However, these visualizers still focus on tracing, which may be more confusing when dealing with recursion [Conversation with Advisor]. Instead, it might be helpful to have the ability to visualize critical steps in a recursive program, rather than visualizing every step [Conversation with Advisor]. As such, we propose a program visualizer which we shall call the abstract program visualizer. The abstract program visualizer would have the ability for users to specify what part of their program they would want visualized [Conversation with Advisor]. This could be implemented in a few different ways. One method is to have users directly specify which lines of codes they would want visualized, which follows from the recommendation from Jeliot’s development team [BABL+11]. Another method is to have the option for users to specify a precondition which when met, would visualize the current state of the program [Conversation with Advisor]. These features could be particularly helpful for instructors when teaching recursive programs, as they could tailor an exercise with only the most relevant visualizations ready in order to help students better understand a recursive program. However, once students have a better understanding of recursion, having the ability to specify which parts of a recursive program students would like visualized would also help increase student engagement with the tool.
8.2 Important Characteristics

Based on the design principles and recommendations of well-supported program visualizers and their developers, the abstract program visualizer should also have the following characteristics:

**Easy-to-Use:** Accessibility was highlighted by both Jeliot and Python Tutor’s developers as important for widespread use [BABL+11, Guo21]. In practice, this would mean having an online implementation, automatic animations, and a simple interface uncluttered by too many configuration settings.

**Complete Visuals:** Another important point emphasized by both Jeliot and Python Tutor’s developers is having the entire state of the notional machine visualized so that the notional machine’s rules are explained by the visuals themselves [BABL+11, POJ+20].

**Engagement:** This principle could be implemented in various ways, with the easiest being the ability to specify questions that would appear during a visualization. Furthermore, having students specify which lines of code to visualize or input preconditions for visualization acts as another form of engagement. Other methods include integrating the tool with online learning materials, or even having the option for users to execute a program themselves in the style of UUhistle [BABL+11, Guo13, Sor12].

**Sustainability:** If the tool is primarily meant as a useful tool for helping students improve their understanding of recursion, then sustainability should be kept in mind throughout the development of the tool. Sustainability is not only important for making the tool realistically maintainable, but also for studying both the long-term benefits of the tool and how to improve it [BABL+11, Guo13].

In order to test this visualizer, a study could be conducted in a CS1 course with a prototype of the tool to see if it helped students learn how to write and understand recursive programs. Ideally in the long term, a study similar to Levy et al. could be conducted with the tool to see its long term
9 Conclusion

In conclusion, inaccurate mental models may be a large contributor to poor understanding of program execution in CS1 courses and beyond. Explicitly teaching a notional machine in CS1 courses seems to be useful in creating more accurate models of program execution, with one of the most popular and successful ways of doing so being through program visualizers. However, it is important to understand what the most effective program visualizers get right and what their developers recommend for future work in order to understand program visualizers and where future work lies. As no program visualizers seems to focus on visualizing recursion, we recommend creating and testing a program visualizer made for this purpose and which implements the recommendations from other program visualizers.

10 References


