Visualizing Environments of Modern Scripting Languages

Kaian Cai[a], Martin Henz[b], Kok-Lim Low[c], Xing Yu Ng[d],
Jing Ren Soh[e], Kyn-Han Tang[f], and Kar Wi Toh[g]
School of Computing, National University of Singapore
caikaian@u.nus.edu,{henz,lowkl}@comp.nus.edu.sg, {e0726386,e0360572,e0727253,e0544032}@u.nus.edu

Keywords: Learning Programming, Visualization Tools, Integrated Development Environments, Lexical Scoping, First-class Functions

Abstract: A central learning objective of introductory programming courses is a thorough understanding of environments that arise when programs written in modern programming languages run. An awareness is arising in the CS-Ed community that a mental model based on a runtime stack does not do justice to languages that combine lexical scoping with first-class functions. As a result, debugging and visualization tools designed around a runtime stack are not suitable for this family of languages, which includes Python, JavaScript, Ruby, Lua, Java, and Scheme. As a suitable mental model for environments in these languages, the classical programming textbook "Structure and Interpretation of Computer Programs" (SICP) introduced the *environment model of computation* using diagrammatic graphics. The SICP authors Hal Abelson and Gerald Jay Sussman designed the environment model to represent the runtime data structures required for executing programs written in such languages while blending out all forms of control. In this paper, we describe a novel tool for automatically and interactively visualizing the execution environments of programs written in the targeted language family. After introducing the environment model in detail, we highlight the main challenges for its automatic and interactive visualization. We outline the architecture of the tool and its integration into a web-based environment for learning the structure and interpretation of computer programs and conclude with an analysis of the tool's impact based on feedback from 69 course facilitators in Academic Year 2021/22.

1 INTRODUCTION

Environment structures evolve during the interpretation of programs written in languages with lexical scoping and first-class functions, such as all modern scripting languages (Python, JavaScript, Ruby, Lua, etc). A thorough understanding of environment structures is typically a central learning objective of an introductory computer science course that uses such a language. The textbook *Structure and Interpretation of Computer Programs* (SICP, (Abelson and Sussman, 1996)) introduced the *environment model* as a pedagogical tool. Chapter 3 of SICP explains the application of functions in programs of lexically-scoped languages with first-class functions informally, followed by an implementation in a metacircular evaluator in Chapter 4. The application of functions plays a central role; SICP Section 3.2.1 gives the following high-level explanation.

> To evaluate a [function] application, evaluate the subexpressions of the application, [and then] apply the value of the [function] subexpression to the values of the argument subexpressions.

The evaluation of expressions in programs of these languages must be carried out with respect to an *environment* that keeps track of the values that program names refer to. Assignment to a name is then explained as an operation that destructively changes the value that the name refers to in the environment.

While teaching a first-semester computer science course using the JavaScript adaptation of SICP (Abelson and Sussman, 2022) (SICP JS) at the National University of Singapore (NUS), we saw the need for a visualization tool that illustrates the evaluation of programs in the presence of lexical scoping and first-class functions. A review of recent related work in Sec-

[a] https://orcid.org/0000-0003-3130-275X
[b] https://orcid.org/0000–0002-6529-5896
[c] https://orcid.org/0000–0001-6837-5835
[d] https://orcid.org/0000-0001-6954-0126
[e] https://orcid.org/0000-0002-0677-2605
[f] https://orcid.org/0000-0002-9396-2895
[g] https://orcid.org/0000-0001-6564-4235

tion 2 reveals that in current tools for visualizing the execution of programs, a runtime stack plays a central role. We argue that environment structures in our target languages do not follow a stack discipline, and therefore conventional stack-based debugging and visualization tools are not suitable in the context of our course. In response to our teaching needs, we have developed a novel environment model visualization tool and integrated it in Source Academy (Anderson et al., 2023), a web-based environment for learning the structure and interpretation of computer programs. Section 3 reviews the environment model in detail, using examples from SICP Chapter 3. Section 4 shows how our visualizer displays the environments of example programs. Section 5 describes the main implementation challenges for interactive visualization of environments. In Section 6, we outline the architecture of the tool and its integration into Source Academy. In Section 7, we provide a quantitative analysis of the benefits of using an environment model visualizer, based on a survey of 69 tutors who used the tool in Academic Year 2021/22.

2 RELATED WORK

Integrated development environments typically provide extensive debugging tools for analyzing the runtime behavior of programs; examples are Microsoft Visual Studio (Microsoft, 2023) for a wide variety of programming languages, and IntelliJ IDEA (JetBrains, 2023) for Java, Kotlin, Groovy and other languages whose implementations use the Java Virtual Machine. Leading web browsers such as Google Chrome and Mozilla Firefox provide support for debugging JavaScript programs (Google, 2023; Mozilla, 2023). Python Tutor (Guo, 2021) provides a visualization of the execution of Python programs. These tools are based on a runtime control stack as their central mental model. Clements and Krishnamurthi point out the limitations of this approach for the target language family in (Clements and Krishnamurthi, 2022):

> The premise of this paper is that the traditional depiction of the stack, as a sequence of self-contained frames, is insufficient and potentially misleading, mirroring and even creating troubling misconceptions.

Indeed, none of these tools clearly explains the relationships between environment frames and function values (closures) that unfold when programs use functions a first-class values. Therefore using them in a large first-year course that aims to cover first-class functions would be of questionable merit. Instructors and facilitators would need to spend precious time on explaining the differences between a suitable model for environments and the representations in the tool. Clements and Krishnamurthi (Clements and Krishnamurthi, 2022) don't provide a tool for visualizing environments but study alternative pedagogical approaches using Snap! blocks (Harvey and Mönig, 2010) that are manually arranged by learners.

Related to this work is the concept of a *notional machine*, which is an abstraction of a computer that explains the execution of programs as a step-by-step process. In his dissertation (Sorva, 2012), Sorva points out that learners inevitably construct notional machines for program execution and argues that they benefit from being deliberately introduced to pedagogically sound notional machines. The dissertation and (Sorva, 2013) contain in-depth discussions of notional machines, program visualization tools, and their pedagogical value, but do not cover lexically scoped languages with first-class functions.

The substitution model of SICP Chapter 1 can be seen as a notional machine for program execution, and interactive tools such as the Racket Stepper (Racket team, 2021) and the Source Academy stepper (Henz et al., 2021) can be seen as interactive visualizations of such a notional machine. Substitution-based program visualization tools are limited to purely functional programs (no assignment) and provide a rather low level of abstraction.

Related to the visualization of data structures covered in Section 4 is the Racket data visualizer (Rosenthal, 2019) and the IntelliJ plugin Java Visualizer (Lipsitz, 2019), but these tools are not designed to present their data visualization in the context of environments.

Landin's SECD machine (Landin, 1964) introduced environments as a data structure for keep track of name bindings during the interpretation of expressions in the call-by-value lambda calculus. While minimalistic in the covered language features, the SECD machine contains the essential ingredients of the targeted language family and could be extended into a notional machine for this family. Danvy (Danvy, 2004) systematically explores SECD variants, and compilation-based variants have served as inspiration for implementations of functional programming languages and their correctness proofs, most recently in (Kunze et al., 2018).

3 THE ENVIRONMENT MODEL

In the environment model of evaluation, a function is a pair consisting of some code and a pointer to an environment. Functions are created by evaluating a

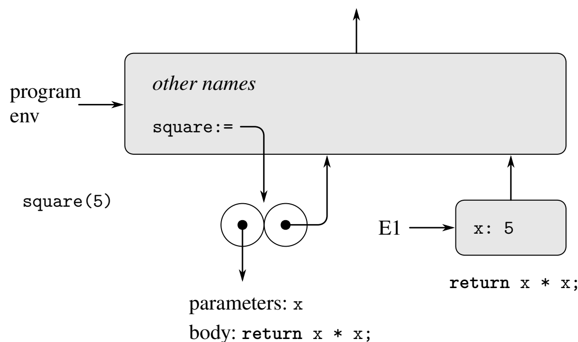

Figure 1: Environment created by evaluating `square(5)` in the program environment. [In the diagrams, `:=` indicates a JavaScript constant and `:` a JavaScript constant.]

lambda expression, which produces a function whose code is obtained from the text of the lambda expression and whose environment is the environment in which the lambda expression was evaluated to produce the function. SICP JS illustrates the concept in Section 3.2.1 using the square function as an example:

> For example, ...
>
> ```
> const square = x => x * x;
> ```
>
> ... evaluates `x => x * x` and binds `square` to the resulting value, all in the program environment. Figure 1 shows the result of evaluating this declaration statement. ... The function object is a pair whose code specifies that the function has one parameter, namely `x`, and a function body `return x * x;`. The environment part of the function is a pointer to the program environment, since that is the environment in which the lambda expression was evaluated to produce the function. A new binding, which associates the function object with the name `square`, has been added to the program frame.

Function application leads to the evaluation of the function body in an environment that extends the function's environment with a binding of the parameter `x` to the argument 5, as depicted in Figure 1. Using a `make_withdraw` function

```
function make_withdraw(balance) {
  return amount => {
      if (balance >= amount) {
        balance = balance - amount;
        return balance;
      } else {
        return "insufficient "
               + "funds";
      }
   };
}
```

SICP JS explains the evaluation of programs with local state in Section 3.2.3 as follows.

> The interesting part of the computation happens when we apply the function `make_withdraw` to an argument:
>
> ```
> const W1 = make_withdraw(100);
> ```
>
> We begin, as usual, by setting up an environment E1 in which the parameter `balance` is bound to the argument 100. Within this environment, we evaluate the body of `make_withdraw`, namely the return statement whose return expression is a lambda expression. The evaluation of this lambda expression constructs a new function object, whose code is as specified by the lambda expression and whose environment is E1, the environment in which the lambda expression was evaluated to produce the function...

The application of account `W1` to the amount 50

```
W1(50);
// result: 50
```

leads to the environment structure of Figure 2. Note that the application of `W1` in the program environment leads to an extension of E1, which at that point is only accessible through the function object to which `W1` refers. The example "Closures Outlive Stack Frames" in Section 4 of (Clements and Krishnamurthi, 2022) mirrors this example to illustrate that environments do not follow a stack discipline. The first frame of E1 outlives the function call that created it and serves as enclosing environment for calls of `W1`.

The environment model as presented here is *minimal* in the sense that all components of runtime structures are forced to be present by the semantics of the language. Lexical scoping forces environment frames to refer to their parent frames, and function values (closures) to carry the environment in which they were created.

4 THE VISUALIZATION TOOL

Figure 3 shows how the visualizer displays the environments of Figure 2. The visualizer is integrated in Source Academy (Anderson et al., 2023), a web-based system for supporting the teaching of courses that use SICP JS. The learners enter their programs in the editor on the left. Before program execution, a "Run" button in place of "Resume"—indicated by Arrow 1—lets learners execute their programs and view the results. Note the circle in the beginning of

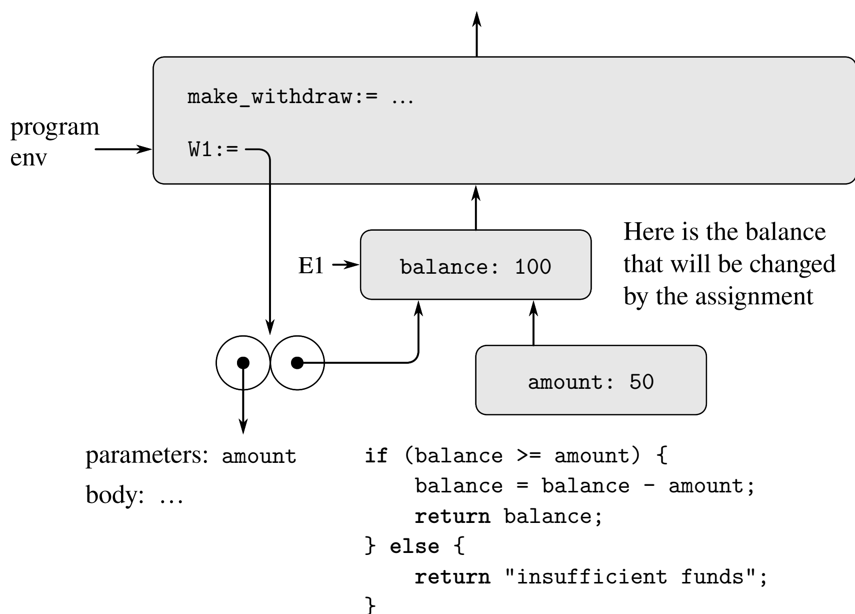

Figure 2: Environments created by applying the function object W1.

line 11, which represents the breakpoint added by the learner to pause program execution just before the line is reached. When a program encounters a breakpoint, the environment visualizer displays the environment structures that have been created before the line in which the breakpoint appears. The "Run" button turns into a "Resume" button. A "Stop Debugger" button aborts program execution, indicated by Arrow 2.

The runtime representation of data structures typically poses challenges to learners, especially in the presence of first-class functions. Figure 4 shows how the environment model visualizer displays the application of a `map` function to a three-element list, which is made up of pairs whose components are accessed with the functions `head` and `tail`.

```
const square = x => x * x;
const map =
(f, xs) => is_null(xs)
           ? null
           : pair(f(head(xs)),
                  map(f, tail(xs)));
const ys = map(square, list(2,3,4);
```

The figure shows the situation after pressing "Resume" once, so we see the state at the beginning of the first recursive call of `map`.

The tool is integrated in the latest release of Source Academy (Source Academy, 2023a), which also contains an interactive version of SICP JS. Both are available in the repositories of the Source Academy GitHub organization (Source Academy, 2023d) under free licences.

5 IMPLEMENTATION CHALLENGES

The most obvious implementation challenge is posed by the browser-based architecture of Source Academy. The usual implementation pathway for JavaScript programs in a browser-based architecture consists of passing the learner's JavaScript programs to the browser's JavaScript engine. As described in (Anderson et al., 2021), Source Academy uses a JavaScript parser to restrict the JavaScript sub-language available to the learners, and a JavaScript-to-JavaScript transpiler that improves error messages and realizes proper tail calls. The challenge for implementing the environment model visualizer in such a system is the lack of access to the browser's internal data structures for representing environments and to the browser's runtime environment in order to interactively pause and resume program execution as a result of setting breakpoints in the program. We solved these problems by implementing an alternative implementation pathway for JavaScript programs that uses an interpreter instead of a transpiler. The sub-languages approach of the course and Source Academy facilitates this solution, because only those aspects of JavaScript that are relevant for SICP JS need to be implemented by this interpreter.

The second challenge is posed by the need to interactively pause program execution at any breakpoint, display the environment data structures that have accumulated up to the breakpoint and resume evaluation upon pressing "Resume". This is handled by perva-

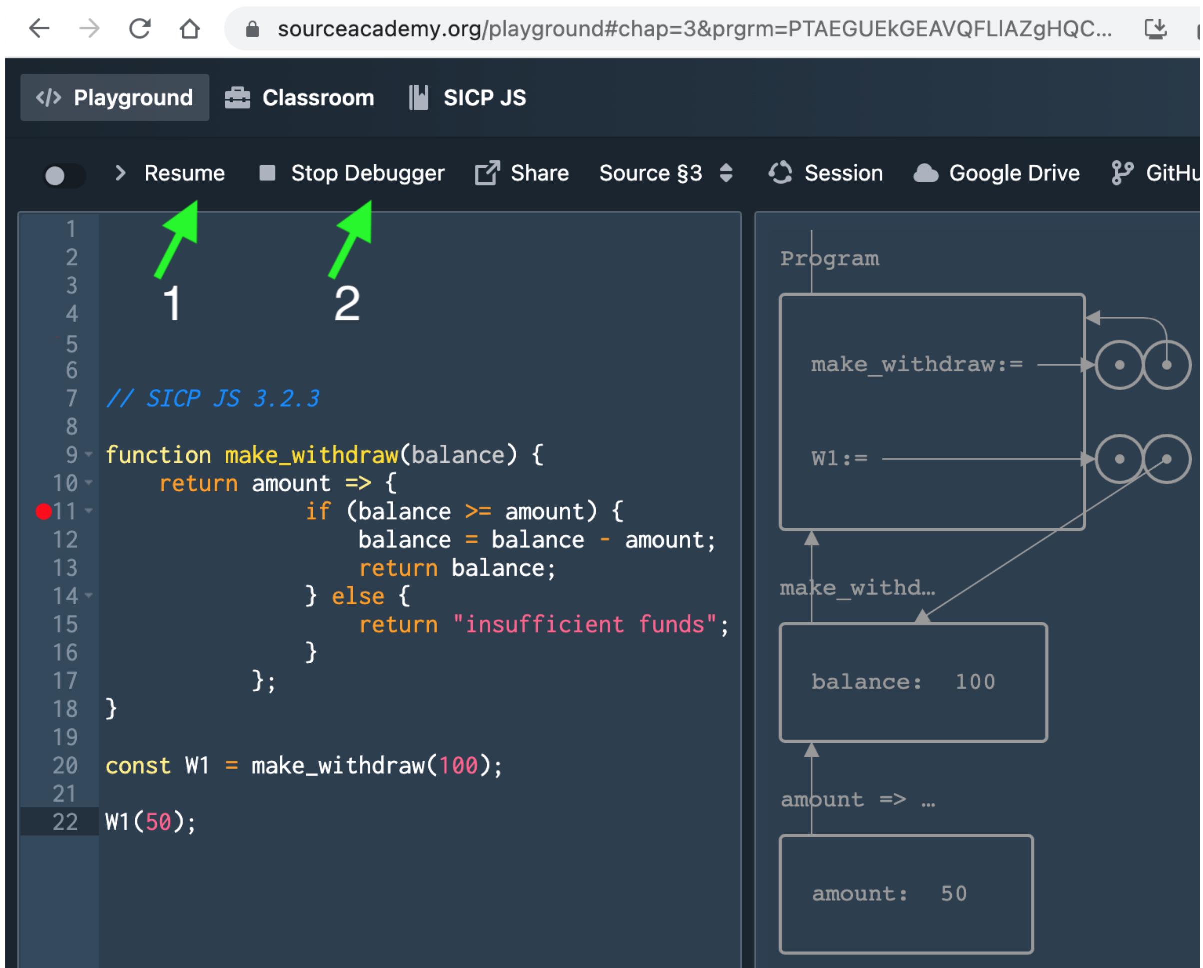

Figure 3: Environment model visualizer, showing the environments of Figure 2.

sively using asynchronous JavaScript functions in the implementation of the interpreter. An alternative solution is presented in (Baxter et al., 2018) which would have led to transforming the mentioned transpiler into continuation-passing style. We rejected this option to keep the transpiler as simple as possible. The interpretation overhead is not critical because the visualized programs are typically short-running.

A third implementation challenge is posed by a pedagogical aspect of the environment model as presented in SICP. According to the environment model diagrams in SICP, environment frames do not “expire”. Once created, they persist and should be displayed even if they are not reachable after resuming evaluation. It is therefore not sufficient to display the reachable environment frames from the current frame. We explored two options of addressing this challenge. The first option consists of establishing a registry of environment frames to which any new frame is added and that is used for the display. The second option turns environments into a doubly-linked data structure: Parent frames include references to their child frames, in addition to the usual parent pointers in child frames. We chose the second option, because the visualization can naturally traverse and display the environment structure starting from the global frame. Both options obviously keep all environment frames alive in the heap of the JavaScript runtime system. This does not pose performance problems because the environment model visualizer is used only for quite simple programs.

An important functional requirement is consistency across breakpoints. Once a frame has been drawn for one breakpoint, its position should not change when it gets displayed again for further breakpoints so that the learner recognizes previously seen frames by their position on the screen. This constraint does not allow the visualizer to recompute a globally optimized layout for environments at each breakpoint.

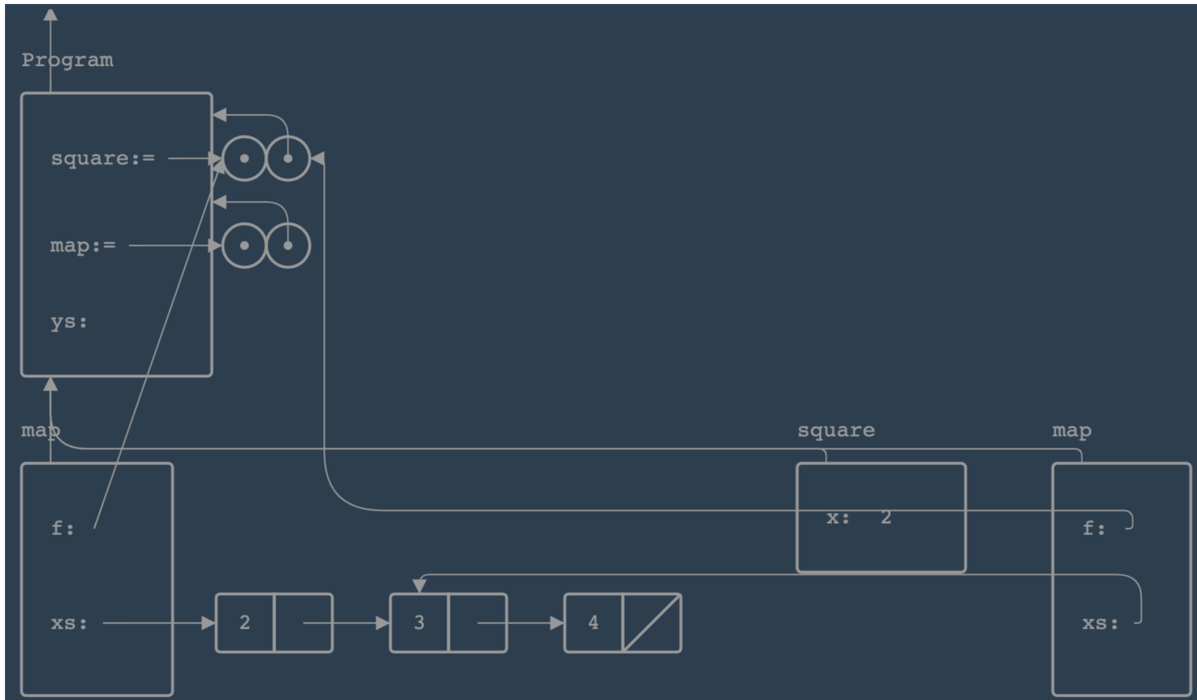

Figure 4: Environment model visualizer, showing the environments at the beginning of the first recursive call of a `map` function.

6 ARCHITECTURE OF THE VISUALIZATION TOOL

The visualization tool is integrated into the Source Academy (Anderson et al., 2023), a browser-based environment for entry-level programming. The system is designed and built by students and staff of NUS, is published under an Apache license (Source Academy, 2023b), and is implemented as a React application (Meta Platforms, Inc., 2023). Source Academy switches to the JavaScript interpreter described in the previous section when at least one breakpoint is set. Setting a breakpoint inserts a **`debugger;`** statement in the program.

The interpreter is implemented—along with all other JavaScript-specific system features—in a separate Node.js package for browser-independent development and testing (Source Academy, 2023c). Whenever the interpreter meets a **`debugger;`** statement, the current program context is sent to the visualizer. The visualizer receives the context in the form of the doubly-linked environment data structure described in the previous section, which represents all frames created so far during the execution of the program. The data structure is rooted at the global frame.

The visualization is implemented with Konva.js (Lavrenov, 2023). A `Layout` class encapsulates the canvas and layout calculations. The singleton `Layout` instance contains a `Grid`, which in turn contains an array of Levels containing frames, pairs, and function objects, each with their (x, y) coordinates on the canvas. We investigated published graph drawing algorithms (for an overview, see (Battista et al., 1998)) and several publicly available graph drawing frameworks, but decided to implement the layout from scratch to provide predictable and visually pleasing diagrams and meet the consistency requirement given at the end of Section 5.

7 RESULTS

The environment visualizer was designed and developed over the past five years by undergraduate students at NUS, following the community-based development approach described in (Anderson et al., 2023), and previous versions were integrated into Source Academy. The instructors of the SICP-based course at NUS decided that the visualizer needs to be 100% consistent with the textbook, and this consistency was only achieved in 2021. As a result, 2021 was the first year in which the visualizer was used in class and the students were encouraged to used it while learning the environment model.

The 2021 course had 667 students and a teaching team of close to 100 staff. Three lecturers man-

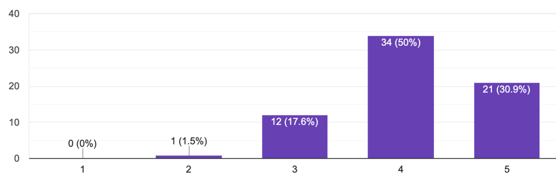

Figure 5: Results of survey question: "The environment model visualizer [...] helped my students in understanding the environment model of SICP JS. (1 for strongly disagree, 5 for strongly agree)". Of the 69 corespondents, 68 answered this question.

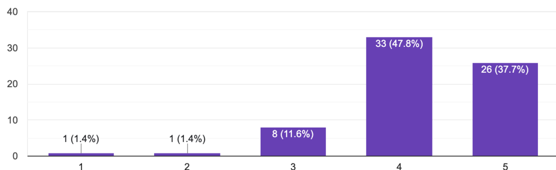

Figure 6: Results of survey question: "The environment model visualizer [...] accelerates the learning process by letting students interactively explore the the environment structures that their programs give rise to. (1 for strongly disagree, 5 for strongly agree)"

aged the course and prepared and delivered the lectures (one two-hour session and one one-hour session in each of the 13 weeks of the semester). The lecturers and six additional tutors (graduate or senior undergraduate students) shared the facilitation of a weekly one-hour recitation session, conducted in groups of 20 to 24 students. A team of 88 undergraduate assistants facilitated weekly 2-hour small-group lab sessions with maximally 8 students per group.

The utility of the visualizer was obvious to the teaching staff of the course, and therefore A/B testing was deemed unethical. Instead, to assess the degree in which the visualizer contributed to the learning and teaching of the course, we conducted an anonymous survey among the 88 undergraduate assistants and the 6 tutors. From these 94 facilitators, we received 69 responses (73%).

The first two survey questions asked the facilitators about the degree in which the visualization tool contributed to the learning of their students. Figure 5 shows their responses regarding the overall efficacy of the visualizer, i.e. whether it helped their students in understanding the environment model. Of the responding facilitators, 80.9% agreed or strongly agreed that it did, and only one disagreed. The second question more specifically addressed the speed of the learning process and the interactive features of the visualizer. Figure 6 shows that 85.5% agreed or strongly agreed that the visualizer accelerated the learning process, one disagreed and one strongly disagreed. From these responses, we conclude that the visualizer is considered by facilitators to be an effective tool for learning the environment model of computation.

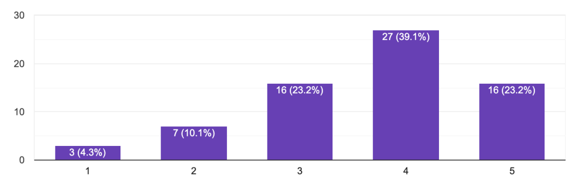

Figure 7: Results of survey question: "The environment model visualizer [...] helped me as instructor to explain the environment model. (1 for strongly disagree, 5 for strongly agree)"

The last survey question asked the facilitators whether the tools helped them explain the model. Figure 7 shows that 62.3% agree or strongly agree that the visualizer helped them explaining the environment model in their sessions, whereas 14.4% disagree or strongly disagree. From these responses, we conclude that the visualizer is an effective classroom tool for teaching the environment model of computation.

8 CONCLUSION AND FUTURE WORK

Starting from the need for a visual representation of runtime structures to facilitate the teaching of first-year computer science courses, we reviewed the environment model of SICP. We argued that the model is minimal in the sense that all its components are forced by the semantics of lexically scoped languages with first-class functions. We described an interactive tool for visualizing SICP's environment model and its implementation in Source Academy, an environment designed for supporting the teaching of SICP-based courses. To our knowledge it is the first interactive tool that graphically represents the environment structures that arise during the interpretation of Scheme, JavaScript, and by extension other languages that share the central characteristics of lexical scoping and first-class functions, including Python, Ruby, and Lua.

The environment model as presented in SICP (Abelson and Sussman, 1996) deliberately excludes the control aspect of program execution in order to keep the model simple enough for an introductory textbook. Thus the environment model and by extension our environment model visualizer do not qualify as a notional machine in the sense of (Sorva, 2013), who argues conclusively that students benefit from explicit notional machines and their interactive graphical visualization. In future work, we will approach the notional-machine challenge posed

by (Clements and Krishnamurthi, 2022) for the targeted language family with the environment model of SICP as the starting point. For this, we will extend the environment model with control components to obtain a proper notional machine for languages with lexical scoping and first-class functions. We hope that this machine will put us in a position to evolve our environment model visualizer into a complete, interactive notional machine visualizer: a tool for the detailed execution visualization of programs written in the targeted language family.

REFERENCES

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd edition. With Julie Sussman.

Abelson, H. and Sussman, G. J. (2022). *Structure and Interpretation of Computer Programs, JavaScript edition*. MIT Press, Cambridge, MA. adapted to JavaScript by Martin Henz and Tobias Wrigstad, with Julie Sussman.

Anderson, B., Henz, M., and Low, K.-L. (2023). Community-driven course and tool development for CS1. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education*, page 7, Toronto, Canada. ACM.

Anderson, B., Henz, M., Low, K.-L., and Tan, D. (2021). Shrinking JavaScript for CS1. In *Proceedings of the 2021 ACM SIG-PLAN SPLASH-E Symposium (SPLASH-E '21)*, pages 87–96, Chicago, IL. ACM.

Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. (1998). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Saddle River, NJ.

Baxter, S., Nigam, R., Politz, J. G., Krishnamurthi, S., and Guha, A. (2018). Putting in all the stops: execution control for JavaScript. In *PLDI 2018: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–45, Philadelphia PA. ACM.

Clements, J. and Krishnamurthi, S. (2022). Towards a notional machine for runtime stacks and scope: When stacks don't stack up. In *ICER '22: Proceedings of the 2022 ACM Conference on International Computing Education Research*, pages 206–222. ACM SIGCSE.

Danvy, O. (2004). A rational deconstruction of Landin's SECD machine. In Grelck, C., Huch, F., Michaelson, G., and Trinder, P. W., editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, volume 3474 of *LNCS*, pages 52–71, New York. Springer.

Google (2023). Chrome DevTools. https://developer.chrome.com/docs/devtools.

Guo, P. (2021). Ten million users and ten years later: Python Tutor's design guidelines for building scalable and sustainable research software in academia. In *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology*, page 1235–1251, New York. Association for Computing Machinery.

Harvey, B. and Mönig, J. (2010). Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? In *Proceedings of Constructionism 2010*, pages 1–10, Bratislava, Slovakia. Library and Publishing Centre, Faculty of Mathematics, Physics and Informatics, Comenius University.

Henz, M., Tan, T., Chua, Z., Jung, P., Tan, Y.-J., Zhang, X., and Zhao, J. (2021). A stepper for a functional JavaScript sublanguage. In *Proceedings of the 2021 ACM SIG-PLAN SPLASH-E Symposium (SPLASH-E '21)*, pages 71–81, Chicago, IL. ACM.

JetBrains (2023). IntelliJ IDEA. https://www.jetbrains.com/idea.

Kunze, F., Smolka, G., and Forster, Y. (2018). Formal small-step verification of a call-by-value lambda calculus machine. In Ryu, S., editor, *Asian Symposium on Programming Languages and Systems, LNCS, volume 11275*, pages 264–283, Cham, Switzerland. Springer International Publishing.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320.

Lavrenov, A. (2023). Konva.js—an HTML5 Canvas JavaScript framework. https://konvajs.org.

Lipsitz, E. (2019). Java Visualizer—plugin for IntelliJ IDEA. https://github.com/elipsitz/java-visualizer-intellij-plugin.

Meta Platforms, Inc. (2023). React—a JavaScript library for building user interfaces.

Microsoft (2023). Microsoft Visual Studio. https://visualstudio.microsoft.com.

Mozilla (2023). Firefox JS Debugger. https://firefox-source-docs.mozilla.org/devtools-user/debugger.

Racket team (2021). The stepper. https://docs.racket-lang.org/stepper.

Rosenthal, J. (2019). Sdraw: Cons-cell diagrams with Pict. https://docs.racket-lang.org/sdraw/index.html.

Sorva, J. (2012). *Visual Program Simulation in Introductory Programming Education*. PhD thesis, Aalto University, Espoo, Finland.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):1–31.

Source Academy (2023a). Public site of Source Academy. https://sourceacademy.org.

Source Academy (2023b). Software repository of frontend. https://github.com/source-academy/frontend.

Source Academy (2023c). Software repository of js-slang. https://github.com/source-academy/js-slang.

Source Academy (2023d). Source Academy GitHub organization. https://github.com/source-academy.