Theia: Automatically Generating Correct Program State Visualizations

Josh Pollock
University of Washington
United States
joshpoll@cs.washington.edu

Jared Roesch
University of Washington
United States
jroesch@cs.washington.edu

Doug Woos
Brown University
United States
doug_woos@brown.edu

Zachary Tatlock
University of Washington
United States
ztatlock@cs.washington.edu

Abstract

Program state visualizations (PSVs) help programmers understand hidden program state like objects, references, and closures. Unfortunately, existing PSV tools do not support custom language semantics, which educators often use to introduce programming languages gradually. They also fail to visualize key pieces of program state, which can lead to incorrect and confusing visualizations.

Theia, a *generic* PSV framework, uses formal abstract machine definitions to produce *complete*, *continuous*, and *consistent* (CCC) PSVs.

To produce CCC visualizations with Theia, an educator only needs to specify an abstract machine and optionally customize the resulting web page, allowing her to visualize custom language semantics without developing a language-specific tool.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Keywords abstract machine, notional machine, program visualization, CS1, CS2, operational semantics

ACM Reference Format:
Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: Automatically Generating Correct Program State Visualizations. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3358711.3361625

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10.
https://doi.org/10.1145/3358711.3361625

1 Introduction

Novices and experts alike struggle to understand the runtime states of their programs. Students find the most difficulty with *hidden state*, like objects, closures, and references, that do not have syntactic representations [Sorva 2013].

Program state visualization (PSV) tools, such as Python Tutor [Guo 2013], Jeliot [Moreno et al. 2004], and Novis [Berry and Kölling 2016], help students understand hidden state by visualizing program traces. But these tools only support a small set of languages and may produce misleading PSVs of the languages they do visualize (Figure 1). As a result, educators often resort to crafting PSVs by hand and are unable to give automated tools to their students (Section 2).

In this paper we present Theia, a tool for creating PSVs *automatically* from an abstract machine definition. Theia allows instructors to quickly make new visualizations of program executions and gracefully grow them as the set of semantic features expands throughout a course. It uses formal abstract machines expressed in the K framework [Roşu and Şerbănuţă 2010] to automatically provide visualizations for any instructor-defined language semantics (Section 3).

Based on prior work, we identify three key goals for novice-oriented PSV tools. Such a tool should be:

(1) *complete.*[1] It should visualize the entirety of each intermediate program state.
(2) *continuous.*[1] It should visualize transitions between program states.
(3) *consistent.*[2] It should visualize similar constructs across languages similarly.

We refer to these criteria collectively as *CCC*. Though in general, students and instructors may find it useful to summarize and customize visualizations in ways that violate these principles, we believe CCC is essential for a first introduction to language semantics. Completeness and continuity ensure all semantic information is present in the PSVs, and consistency means a student can learn new semantic features using a small, fixed visual vocabulary.

[1] [Levy et al. 2003]
[2] [Dragon and Dickson 2016]

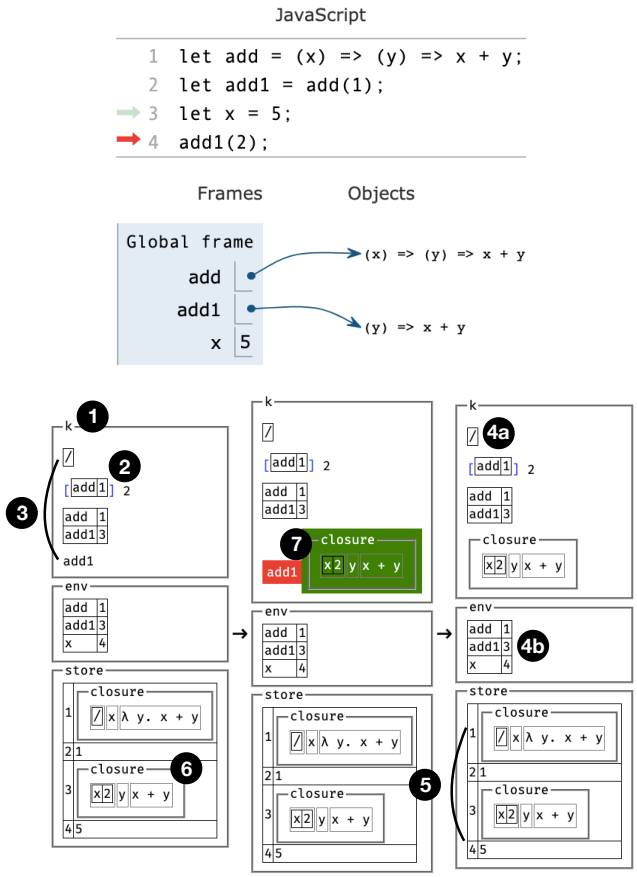

Figure 1. Python Tutor (top) does not visualize closures completely, since it omits their saved environments. This may lead a student to believe that the free variable x in add1 is bound to 5 instead of 1. Theia (bottom), with a similar program in a functional language, visualizes closures completely. It is labeled with the primitives the Theia prototype supports (see Section 4): ❶ cell, ❷ evaluation context, ❸ expression sequence, ❹a empty environment, ❹b non-empty environment, ❺ store/heap, ❻ boxed value. ❼ is the difference between the two program states on either side. Note: These visualizations were modified to conserve space.

Theia achieves **completeness** through the K framework backend, which provides full information about runtime program state (Section 3); **continuity** by visualizing one *transition* at a time whereas most existing tools visualize one state at a time (Section 3); and **consistency** by decomposing program state into nearly-universal primitives that are visualized identically across languages (Section 4).

Figure 1 provides an example of the Theia prototype's visualizations and demonstrates how a violation of CCC can mislead novices. We illustrate how Theia generates visualizations that are CCC *by construction* through a series of case studies that grow a functional language and visualize an imperative one (Section 5).

Theia's design is guided by Berry's insight that *formalized notional machines are abstract machines* [Berry 1990]. We believe this connection is underappreciated in the literature, because many researchers in the CS Education Research (CER) and Programming Language (PL) communities are unfamiliar with abstract machines (AMs) and notional machines (NMs), respectively. Attempting to narrow the gap between the two communities, we describe NMs and AMs in Section 2, highlighting their similarities. We then examine the potential for NMs in PL and for AMs in CER in Section 7.

Theia serves as a first prototype for a larger research program to build highly extensible program visualizers that instructors can evolve flexibly to help students understand hidden program state (Section 8). We discuss existing approaches to generic visualizations, CCC, and PSVs in Section 6. Theia is publicly available at github.com/uwplse/theia and hosted at theia.software.

2 Background

In this section we survey existing handcrafted visualizations and discuss their strengths and design tradeoffs (Section 2.1). We then provide definitions and examples of notional machines (NMs) and abstract machines (AMs) (Sections 2.2 and 2.3).

2.1 Handcrafted Visualizations

When instructors are not satisfied with automated PSV tools, they often create visualizations manually. We briefly describe techniques instructors use to create handcrafted visualizations. We focus on techniques for ML-like languages using environment-based semantics. Few automated visualization tools support these semantics, which leads instructors to manually construct visualizations.

Figure 2 shows three common ways instructors currently create PSVs: handwritten notes, slideware art, and inline text. Handwritten notes are quick to produce, relatively flexible, and easy for students to replicate; however, they can be difficult to read and animate. Instructors may also find that handwritten traces do not match the polish of the rest of their slides, and thus opt for a different method such as slideware art. Creating custom slide graphics is time-consuming, but provides more clarity than handwritten notes, because slideware allows for animations and more precise, uniform diagrams. Inline text offers a middle ground between the previous two strategies. Creating these visualizations takes less time than slideware while achieving nearly the same polish. Unfortunately, inline text only supports a limited range of PSVs and may confuse students, because it does not clearly distinguish between visualization and source code. All three approaches require significant manual effort *per example* to create clear, useful diagrams. Theia provides an automated alternative aimed to scale effort *per language*.

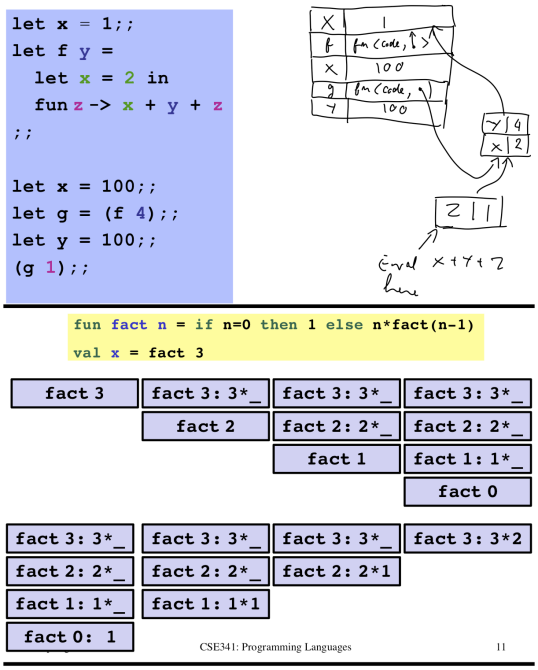

Figure 2. Slides from UC San Diego's CSE 130, University of Washington's CSE 341, and Cornell University's CS 3110, respectively. They are examples of, resp., handwritten notes, slideware art, and inline text.

2.2 Notional Machines

A notional machine (NM) is a construct in CER that describes the machine a student must learn to manipulate with programs. In other words:

> [A NM] is an idealized abstraction of computer hardware and other aspects of the runtime environment of programs… [that] serves the purpose of understanding what happens during program execution. [Sorva 2013]

In CER, NMs are used to compare programming paradigms [Sajaniemi and Kuittinen 2008], identify mental models within a single paradigm [Schulte and Bennedsen 2006; Sorva 2013], or justify the construction of a PSV [Berry and Kölling 2016].

Though NMs serve an important role in these analyses, they are rarely fully described. NM descriptions frequently gloss over subtleties of state *transitions* and instead focus solely on the components of the state [Krishnamurthi and Fisler 2019]. While this approach may be acceptable for coarse-grained analysis, it is too vague for finer-grained comparisons of language choices within the same paradigm, e.g., between call-by-value and call-by-name and between semantically similar languages like C# and Java; Ruby and Python; and F# and OCaml.

Gries and Gries provide one of the most complete descriptions of a NM. They describe two pedagogical Java memory models including both state and transition rules. Their explanations, however, remain informal since they are presented in the form of instructions for a student handwriting a program trace. This conflates the language semantics with the visualization. One must reason about the underlying NM as well as the visualizations and metaphors used to communicate it. When the descriptions are combined, this reasoning becomes unnecessarily difficult.

2.3 Abstract Machines

An abstract machine (AM) is a common construct in PL research. An AM for a language, L, is a state machine (not necessarily finite) with input language L. A formal description of an AM comprises a machine language, L; a description of the AM's state (e.g., a program counter, stack, and heap); and transition rules, which use instructions in L to manipulate the AM's state. AMs are frequently designed to facilitate formal reasoning or language implementation. They are not limited to concrete execution, but also play a useful role in specifying abstract semantics used in abstract interpretation. AM transition rules are often defined using *operational semantics*.[1]

Comparing the definitions of NMs and AMs, one may wonder how they are related. Berry answered this question nearly 30 years ago in his dissertation, where he describes how to generate program animations from formal semantics:

> The abstract machine of the operational semanticist is similar to the notional machines used to teach programming.… The difference is one of abstraction; teachers and students want intuitive designs that illustrate the main points of a design, while operational semanticists want precise and detailed definitions. Thus a notional machine is a simple view of a semantic abstract machine. [Berry 1990]

In other words, while both NMs and AMs attempt to capture abstract models of a machine, they have different goals. Educators and education researchers want machines that aid learning, while programming language engineers and researchers want machines with precise semantics, useful for automated analysis and proof. We argue in Section 7 that this gulf, while natural, is unnecessary and discourages new research questions and discoveries.

[1] See http://pages.di.unipi.it/corradini/Didattica/PR2-B-14/OpSem.pdf for an introduction to operational semantics.

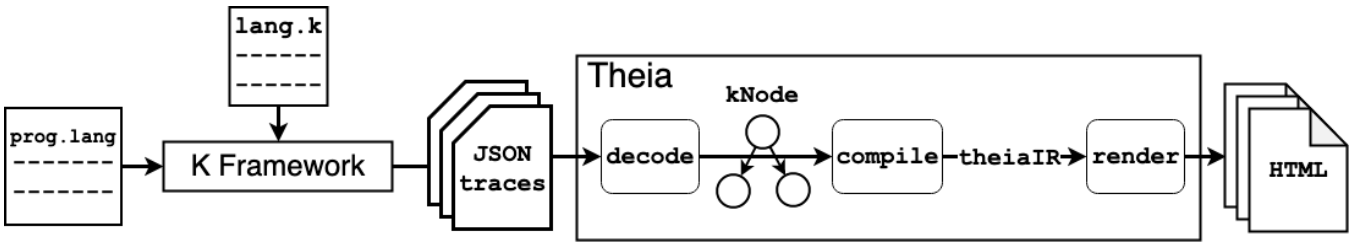

Figure 3. An overview of Theia's architecture with the K Framework backend. The user provides a formal language semantics in K (written once, usu. by an instructor) and a program in that language. Theia visualizes the execution trace of her program.

3 Architecture

In this section we discuss Theia's architecture, and explain how our tool achieves completeness and continuity.

Figure 3 provides an overview of Theia's architecture. Theia is structured as a compiler. It has an intermediate representation (IR) and currently supports a single debugger backend—the K framework—and a single visual frontend—HTML.

The K Framework provides a state logger that produces JSON encodings of ASTs representing each state of a program's execution. In Theia's current implementation, programs are assumed to terminate and all states are logged ahead of time. Theia decodes these JSON files into an initial `kNode` AST that is close to K's own AST. This representation does not distinguish between some components that are visually separated in the final output such as expressions and complex values. Theia performs a compilation pass to separate and reorganize these components, producing `theiaIR`, which is designed to support Theia's visualizations.

Theia's visualizations are *transition-oriented* rather than state-oriented (see Figure 4). Unlike other generic PSV tools, such as Python Tutor, that present the user with a single state at a time, Theia presents the user with a transition between two states. We visualize this as the before and after states connected by a visual difference between them. This design choice is motivated by several in-person conversations on the strengths and weaknesses of Python Tutor and further corroborated by observations of anonymous users of the site. We found that users often thrash back and forth between isolated states to determine what and how the state changes over time. This is necessary in Python Tutor, because the only continuity the tool provides is an indication of the last line that was executed and the next line to be executed. Theia's ability to show deltas between neighboring states of an execution trace provides **continuity**.

3.1 Backend: K State Logging

The K Framework is a tool for formalizing existing languages and experimenting with new syntax and semantics [Roşu and Şerbănuţă 2010]. Researchers have used K to formalize large subsets of real-world languages including Java, x86-64, C, and JavaScript [Bogdănaş and Roşu 2015; Dasgupta et al. 2019; Hathhorn et al. 2015; Park et al. 2015]. K generates a parser, interpreter, state logger, symbolic executor and many other tools automatically from a K specification. This specification includes a grammar, a configuration describing the runtime state of the program, and expressive rewrite rules. One may think of a K specification as an AM where the grammar describes the machine language, the configuration describes the machine state, and the rewrite rules describe the state transitions. K is a suitable backend for Theia, because it generates a state logger and maps closely to AMs.

Theia uses K's state log functionality to obtain program traces. K's log provides complete information about all intermediate states of the specified AM, which Theia uses to construct **complete** visualizations. The state logger can also track which transition rules were applied, perform symbolic execution, and trace rules nondeterministically. Theia currently only uses information about the intermediate states and only supports the concrete execution of terminating, single-threaded, deterministic programs that do not accept user input. These challenges are not fundamental to Theia's approach, but rather matters of engineering effort.

3.2 Frontend: ReasonReact

Theia is written in ReasonML, a syntactic variant of OCaml, and renders `theiaIR` using ReasonReact, a ReasonML implementation of the React web framework. ReasonML is a powerful functional language well-suited to building a compiler that also provides straightforward web integration. Theia implements visualization deltas with `react-visual-diff`, a library that highlights the difference between two React components. The ability to reuse existing React libraries is an important advantage of using ReasonReact.

4 Visualizations of Language Constructs

Many AMs comprise just a few simple state components. By visualizing each of these components independently and in an AM-agnostic way, Theia provides **consistent** visualizations. These state components include cell (§4.1), evaluation context (§4.2), expression sequence (§4.3), environment (§4.4), store/heap (§4.5), and boxed value (§4.6).

AM states often contain components such as program counters and pointers; however, we did not implement these

in our Theia prototype. Though they are useful, they do not significantly add to the demonstration of the core capabilities of our tool. Instead of program counters our case studies (Section 5) rely on evaluation contexts and expression sequences, and reuse integers instead of a separate pointer type. We leave the implementation of program counters and pointers to future work.

Theia allows users to customize its visualizations. We provide an example customization for evaluation contexts in Section 5.1. For the other components, we describe Theia's default visualizations. In all cases, our design rationale is to be conservative. We borrow from existing, common visualization techniques to create visualizations that are more immediately understandable to those familiar with similar PSVs. This similarity is a form of **consistency** across tools, something that previous general AM visualizations have not prioritized (see Section 6). For visual examples of these components, see Section 5.

4.1 Cell

A K configuration is composed of multiple cells, or complex state components, which models most existing AMs. For example, the SECD abstract machine has four cells: **S**tack, **E**nvironment, **C**ontrol, and **D**ump [Landin 1964]. Its spiritual successor, the CEK abstract machine, has three cells: **C**ontrol, **E**nivronment, and **K**ontinuation [Felleisen and Friedman 1986]. We visualize cells as labeled boxes, because we found this visually distinguished between cells even when there were many of them or they were arbitrarily nested. This is similar to visualizations such as Jeliot 3, which uses labeled regions to separate methods, constants, evaluating expressions, and instances [Moreno et al. 2004].

4.2 Evaluation Context

Researchers use several different visualizations of evaluation contexts. Our prototype implementation of Theia supports two: square brackets as in Danvy and Filinski and underlines as in Pareja-Flores et al. and Whitington and Ridge. The latter papers do not demonstrate nested evaluation contexts, so we opt for a stack of underscores. Evaluation contexts are presented in a different color than black to help them stand out when interspersed with program text.

4.3 Expression Sequence

Evaluation contexts save computations for later, but the K framework uses the `KSequence` AST node to save not just evaluation contexts, but arbitrary data structures as well. In Theia, saved data structures are separated by vertical whitespace. This technique is a generalization of call stack visualizations that present the user with a list of stack frames that grows downwards [Guo 2013; Moreno et al. 2004].

4.4 Environment

An environment is a map between identifiers (variable names) and values. Theia visualizes an environment as a 2-column table. This is a common visual representation [Guo 2013].

4.5 Store/Heap

In the Theia prototype, a heap is also rendered as a 2-column table from identifiers to values. However, the identifiers are references into the heap. This is a tradeoff: using a table is complete, because it presents the value of each pointer to the user, but it is less consistent with existing visualizations that use arrows and an unstructured collection of boxes. These visualizations emphasize that, in contrast to a stack, a heap is a graph with data for nodes and pointers for edges.

4.6 Boxed Value

Visualizing complex values separately from expressions is crucial as it hints to the user when evaluation has stopped and distinguishes between code and data. This subtlety is important for languages that employ macros or runtime values that store code (e.g., closures and objects). Based on existing visualizations of lists, objects, and other compound data structures, Theia visualizes a complex value as a labeled box containing a collection of boxes [Guo 2013; Moreno et al. 2004]. This provides visual separation between the different components of the value and provides strong visual separation of complex values from other types of runtime state.

Theia's visualizations comprise visual primitives inspired by existing PSVs. By visualizing these primitives in a language-agnostic way, Theia provides **consistency** across languages.

5 Case Studies

We evaluate the Theia prototype with a series of case studies. Sections 5.1 through 5.4 explore how to grow a small functional language in Theia while preserving CCC. Section 5.5 shows how Theia's visualizations extend consistently to the imperative paradigm.

To further demonstrate that instructors need only specify a semantics to produce a visualization, our case studies are based on semantics and programs written by K framework developers before the creation of Theia.

5.1 Term Rewriting Lambda Calculus with Arithmetic Expressions

When first introducing students to functional programming, an instructor may wish to begin with a lambda calculus extended with simple arithmetic expressions. The instructor picks a term rewriting semantics, allowing her to delay the introduction of an environment, a store, and closures without sacrificing the expressiveness of the language.

We define such a language in K. For familiarity, we present each language semantics in the case studies as a BNF grammar and operational semantics instead of the K source code itself. We briefly explain how one would specify this language in K. One of the syntax rules for our language is written as

```
syntax Exp ::= Val
             | Exp Exp [strict, left]
             | "(" Exp ")" [bracket]
```

This provides the syntax for expressions. Notice the annotations to the right of the rules. `left` and `bracket` are directives for the parser. `strict` tells K that an expression's subexpressions should be evaluated before it can be used in a transition rule. This annotation also generates evaluation context rules automatically.

One of the transition rules is

```
rule (lambda X:KVar . E:Exp) V:Val => E[V / X]
```

This rule specifies lambda application. Note that, for performance reasons, K's variable substitution module is not implemented as a K specification, meaning it is opaque to Theia. One could write substitution explicitly in K to include them in the visualization. We now provide a formal presentation of this language in BNF and operational semantics:

```
Val ::= KVar
      | λ KVar . Exp
      | Int
      | Bool
```

```
Exp ::= Val
      | Exp Exp
      | ( Exp )
      | Exp * Exp
      | Exp / Exp
      | Exp + Exp
      | Exp <= Exp
```

Lambda Application

$$\frac{(\lambda\, X : \texttt{KVar}\ .\ E : \texttt{Exp})\ (V : \texttt{Val})}{E[V/X]}$$

*

$$\frac{(I_1 : \texttt{Int}) * (I_2 : \texttt{Int})}{I_1 * I_2}$$

/

$$\frac{(I_1 : \texttt{Int}) \,/\, (I_2 : \texttt{Int})}{I_1 / I_2}$$

+

$$\frac{(I_1 : \texttt{Int}) + (I_2 : \texttt{Int})}{I_1 + I_2}$$

<=

$$\frac{(I_1 : \texttt{Int}) <= (I_2 : \texttt{Int})}{I_1 \leq I_2}$$

Figure 4 shows one transition in the execution of a simple arithmetic program written in our first language. It is visualized with two different styles of evaluation contexts based on those found in existing literature and visualization tools [Danvy and Filinski 1989; Pareja-Flores et al. 2007; Whitington and Ridge 2017]. In the first style, square brackets are nested until the innermost subexpression matches a rewrite rule. In the second style, a downward-growing stack of underlines represents this nesting. Each visualization style is specified in about five lines of ReasonML. Theia shows not just the before and after states of the program, but also a visual delta between the two states. In this case, the subexpression `2 * 3` inside the innermost evaluation context is rewritten to `6`. Theia presents the before and after states **completely**. It shows *all* nested evaluation contexts. The delta between these states provides **continuity**.

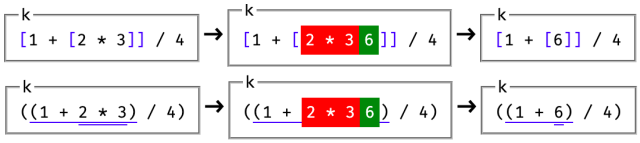

Figure 4. A step in the execution of `(1 + (2 * 3)) / 4` in a rewrite-based lambda calculus presented with two different visual styles for evaluation contexts. Theia allows a user to customize the visualization of each primitive component. The styles above were each specified in about five lines of ReasonML. In this semantics, evaluation contexts are added until a rewrite rule can be applied to the focused subexpression. Notice how Theia visualizes semantics. It is *transition-oriented*, showing before and after states **completely** as well as a visual delta between the two states, which provides **continuity**.

5.2 Switching to an Environment-Based Functional Language (Closures)

After introducing students to a rewrite-based lambda calculus, an instructor may want to switch to an environment-based model. Environments provide an intuitive interpretation of "let" bindings, which may be thought of as adding to a local collection, or *environment*, of bindings. When switching to this model, the instructor must introduce *closures* to preserve the rewrite-based model's lexical scoping rules. Lambdas evaluate to closures, which capture both the lambda and the relevant bindings in the environment where the lambda is defined. Existing visualizations either do not need to visualize closures because they use rewrite-based semantics [Pareja-Flores et al. 2007; Whitington and Ridge 2017] or fail to visualize environments completely [Guo 2013]. In contrast, Theia automatically supports closure visualizations. In our example semantics, a closure is visualized as a boxed value with three components: the saved environment, the name of the lambda's input, and the lambda's body.

The environment-based lambda calculus AM uses multiple pieces of state unlike the rewrite-based one. When the AM state consists of more than one component, K requires the user to define the AM state explicitly and specify the initial position of the source program in that state. The configuration for this language is: `{k: Exp, env: Map, store: Map}` and the program is initially placed in the `k` cell. The `k` cell stores the expression under evaluation, its evaluation contexts, and saved data structures; `env` is a map from variables to pointers; and `store` is a map from pointers to values. We add the following rules to our language, replacing the value syntactic category and lambda transition rules entirely.

```
Val  ::=  closure(MAP, KVar, Exp)

Exp  ::=  ...
       |  if Exp then Exp else Exp    [strict(1)]
       |  let KVar = Exp in Exp
```

Lambda to Closure

$$\frac{\texttt{k: } \lambda\, X : \texttt{KVar}\ .\ E\ \ldots \quad \texttt{env: } \rho}{\texttt{k: closure}(\rho, X, E)}$$

Let

$$\frac{\texttt{let } X = E \texttt{ in } E'}{(\lambda\, X\ .\ E')\ E}$$

If True

$$\frac{\texttt{if true then } E \texttt{ else } \_}{E}$$

If False

$$\frac{\texttt{if false then } \_ \texttt{ else } E}{E}$$

Restore Saved Env.

$$\frac{\texttt{k: } \_ : \texttt{Val} \rightsquigarrow \rho\ \ldots}{\texttt{k: } \_ : \texttt{Val}\ \ldots \quad \texttt{env: } \rho}$$

Variable Lookup

$$\frac{\texttt{k: } X\ \ldots \quad \texttt{env: } \rho(X) = N \quad \texttt{store: } \sigma(N) = V}{\texttt{env: } V\ \ldots}$$

Closure Application

$$\frac{\texttt{k: closure}(\rho, X, E)\ (V : \texttt{Val})\ \ldots \quad \texttt{env: } \rho' \quad \texttt{store: } \sigma}{\begin{array}{l}\texttt{k: } E \rightsquigarrow \rho'\ \ldots \\ \texttt{env: } \rho[X \leftarrow N] \\ \texttt{store: } \sigma[N \leftarrow V] \\ N \text{ is a fresh integer}\end{array}}$$

Note that `strict(1)` means we only require the first argument of `if` to be evaluated before continuing. This allows `if` to short-circuit. $\rightsquigarrow$ denotes a sequenced expression. These sequenced expressions are saved for later, usually for after the first expression has been fully evaluated.

Figure 1 shows that Theia directly supports visualizing multiple cells and complex runtime values like closures. The state is visualized **completely** without any additional effort. Notice the visualization also includes an evaluation context. Since the choice of evaluation context visualization is orthogonal to the rest of the visualization, the instructor only has to pick the visualization once and is guaranteed to have a **consistent** visualization across language semantics.

5.3 Adding Recursive Bindings

The instructor now adds recursive bindings to her language by extending the definitions from the previous subsection:

```
Exp  ::=  ...
       |  letrec KVar KVar = Exp in Exp
       |  mu KVar . Exp
       |  muclosure(Map, Exp)
```

Mu to Muclosure

$$\frac{\begin{array}{l}\texttt{k: mu } X\ .\ E\ \ldots \\ \texttt{env: } \rho \\ \texttt{store: } \sigma\end{array}}{\begin{array}{l}\texttt{k: muclosure}(\rho[X \leftarrow N], E)\ \ldots \\ \texttt{env: } \rho \\ \texttt{store: } \sigma[N \leftarrow \texttt{muclosure}(\rho[X \leftarrow N], E)] \\ N \text{ is a fresh integer}\end{array}}$$

Letrec

$$\frac{\texttt{letrec } F : \texttt{KVar } X = E \texttt{ in } E'}{\texttt{let } F = \texttt{mu } F\ .\ \lambda\, X\ .\ E \texttt{ in } E'}$$

Muclosure Application

$$\frac{\texttt{k: muclosure}(\rho, E)\ \ldots \quad \texttt{env: } \rho'}{\texttt{k: } E \rightsquigarrow \rho'\ \ldots \quad \texttt{env: } \rho}$$

Again the visualization extends without any extra user effort. Notice `muclosures` naturally create a continuation stack by saving continuations with $\rightsquigarrow$. In this semantics, Theia visualizes recursion similarly to Technique #1 in Lewis.

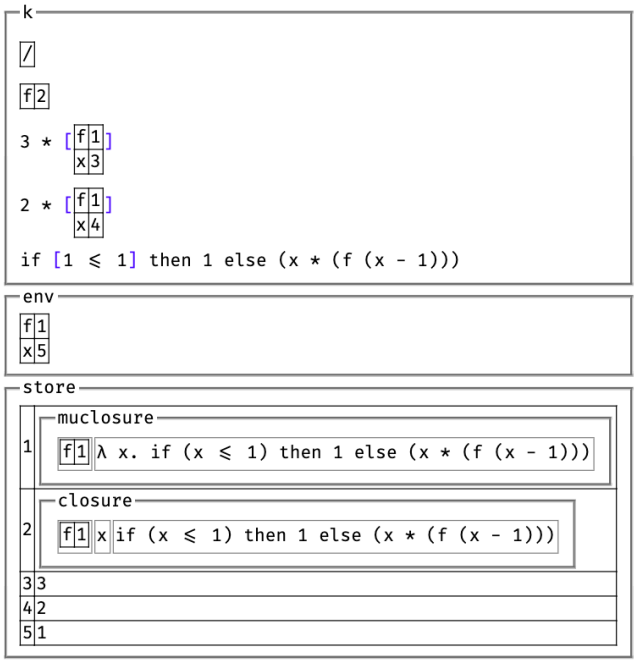

Figure 5. A single state in the execution of `letrec f x = if x <= 1 then 1 else (x * (f (x + -1))) in (f 3)` in an environment-based lambda calculus with `letrec`. Theia creates expression sequences when a semantics saves terms other than evaluation contexts. Here the expression sequence shows recursive calls waiting for values to fill partially evaluated expressions. It also stores old environments, which are restored when the current computation returns. Note: This visualization was modified to conserve space.

5.4 Extending with `callcc`

Finally, the instructor extends her language with `callcc`, a complex control-flow construct, by adding the following

grammar and transition rules. Note that the K Framework provides special semantics for the variable K. It refers to an evaluation context.

```
Exp  ::=  ···
      |   callcc Exp   [strict]

Val  ::=  ···
      |   cc(MAP, K)
```

Save Current Context

```
k: callcc( V : Val) ⇝ K
env: ρ
-------------------------
k: V cc(ρ, K) ⇝ K
env: ρ
```

Restore Context

```
k: cc(ρ, K) V : Val ⇝ _
env: _
-------------------------
k: V ⇝ K
env: ρ
```

Theia's IR is expressive enough to accommodate complicated structures such as `callcc` *without modification*. Code, environments, and evaluation contexts can appear as arbitrary subexpressions in the IR. In contrast, frameworks such as Python Tutor have special support for constructs like closures. Figure 6 shows a Theia visualization of a transition in a program that uses `callcc`.

5.5 An Imperative Language

We began this section by examining functional languages, because few PSV tools support them and their definitions do not require many rules. Although visualizations of functional languages are interesting to study, Theia is not limited to this paradigm. It makes no assumptions about the number or arrangement of cells in an AM configuration nor does it make restrictive assumptions about runtime data structures. To exemplify this, we run Theia on a small imperative language with mutable state and looping constructs.

Figure 7 shows a single transition of an imperative program which sums the numbers 5 to 1 using a while loop. Even though this example is imperative, it still uses an expression sequence and a cell for mapping variables to values. These visual elements are **consistent** with those in the functional languages we visualize above.

6 Related Work

We discuss existing generic tools for AMs that satisfy CCC and existing specialized tools for creating diagrams similar to those already used by teachers and students. We believe Theia is the first visualization framework to provide both CCC and a platform for matching existing visualizations.

6.1 Generic/AM-Based Visualization Frameworks

There are several existing attempts to provide CCC visualizations of abstract machines, though they did not evaluate themselves with those criteria explicitly.

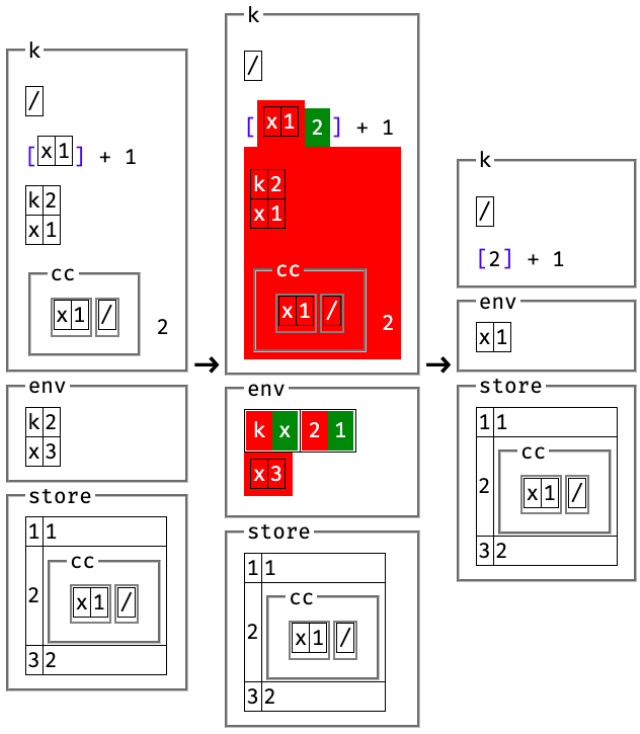

Figure 6. A visualization of `let x = 1 in ((callcc λ k. (let x = 2 in (k x))) + x)`. Theia's visualizations adapt easily to complex constructs like `callcc`. Even though `callcc`'s transitions change several parts of the runtime state, Theia's **continuity** makes the delta visible. `cc`'s saved environment replaces the current one, and its argument is placed in the evaluation context in the expression sequence.

Both PLT Redex [Klein et al. 2012] and the K Framework [Roşu and Şerbănuţă 2010] provide DSLs for specifying operational semantics. To debug these semantics, they also provide visualizations of AM execution. These visualizations, while CCC, are designed for users of their respective tools and thus do not match existing classroom visualizations or allow for easy visualization extensibility. Moreover, the K Framework's visualization tools are currently unmaintained, which prevented a direct comparison with Theia.

GANIMAM [Diehl and Kunze 2000] is a web-based AM visualizer that appears to provide completeness, some form of continuity via interactive animations, and consistency. GANIMAM only supports registers, heaps, stacks, and objects and does not appear to handle evaluation contexts or continuations. We were unable to further evaluate this tool, because it does not seem to be maintained or publicly available.

Berry's thesis discusses the relationship between AMs and NMs and also how to generate program state animations automatically from an operational semantics [Berry 1990]. His work helped inspire Theia's design. We were not able to fully

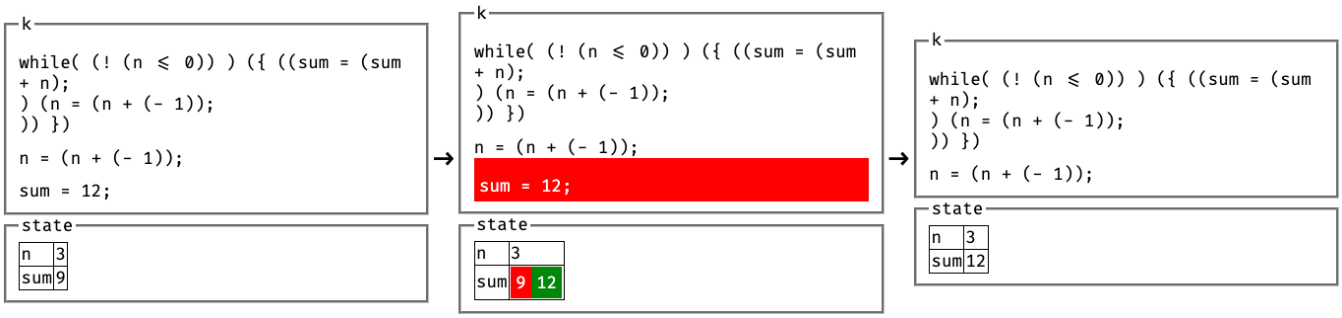

Figure 7. A visualization of the imperative program: `int n, sum; n = 5; sum = 0; while (!(n <= 0)) { sum = sum + n; n = n + -1; }`. Since Theia's visualizations are based on a set of nearly-universal AM primitives, Theia supports multiple paradigms. This example employs cells, an expressions sequence, and an environment mapping variables to values.

Table 1. This table summarizes related work. Theia is the first tool to provide CCC visualizations inspired by common existing PSVs used in classrooms. This should allow Theia to both provide correct visualizations and also, eventually, achieve wider adoption than previous CCC tools. (1) PLT Redex is used by some, but is not widely adopted. (2) Python Tutor supports a few languages, but does not extend to AMs with arbitrary configurations. (3) PLTutor is complete with respect to a reference interpreter, but is not continuous. It supports a single language and so cannot be consistent across languages by default.

Tool	Broad Adoption	Generic	Complete, Continuous, Consistent	Inspired by Popular PSVs
Theia	✗	✓	✓	✓
PLT Redex	—(1)	✓	✓	✗
native K viz	✗	✓	✓	✗
GANIMAM	✗	✓	✓	✗
Berry	✗	✓	✓	✗
Python Tutor	✓	—(2)	✗	✓
PLTutor	✗	✗	—(3)	✓

evaluate Berry's tool, because it does not appear to be publicly available. Though his dissertation contains screenshots of the tool, it is unclear whether it supports data structures like maps or boxed values. Additionally, Berry finds operational semantic steps insufficient and instead introduces *animation steps* to provide more detailed state transitions such as showing steps that focus on subexpressions before performing a rewrite. In contrast, Theia relies only on operational semantics, but achieves similar step granularity to Berry's tool. This is because evaluation contexts can serve the same role as Berry's animation steps, but can be specified as additional rules in an operational semantics. This more fully decouples formal semantics from visualizations.

6.2 Externally Consistent, Language-Specific Tools

Several existing tools emphasize external consistency with existing visualizations or high-fidelity user interactions.

Online Python Tutor is a web-based visualization tool for popular languages like Java, C, C++, JavaScript, and Ruby [Guo 2013]. According to its website, Python Tutor has reached over five million users, making it one of the most widely-used PSV tools. Theia's architecture was partly inspired Python Tutor's, which has an IR for program states and relies on debuggers to produce traces for the visualizer. Unfortunately, while informative for many programs in popular languages, Python Tutor does not visualize constructs such as closures completely (see Figure 1), provides continuity only in the form of a previous and next program counter, and fails to visualize closures consistently across languages. Additionally, Python Tutor's IR only provides *ad hoc* support for closures and other advanced features, making consistently growing languages difficult.[2]

PLTutor [Nelson et al. 2017] is a JavaScript PSV tool intended to be a self-contained language teaching tool. It has instruction-level state transitions, explanations of each transition, and periodic quizzes on program state. A full tutoring system is beyond the scope of Theia. We share PLTutor's creators' belief that a PSV tool must support fine-grained transition rules; however, unlike the creators, we assert that abstract machines are more suitable than implementation-level interpreters for specifying such semantics. Teaching and understanding a language requires multiple semantics and multiple levels of abstraction. Implementation-level interpreters do not provide this flexibility, because they expose

[2] https://git.io/JeYmu describes Python Tutor's execution trace format.

implementation-specific optimizations and representations that are sometimes better ignored.

Finally, several complete and continuous visualizations of functional languages have been developed. We highlight two examples: the Racket Stepper [Findler 2014] and LambdaLab [Sainati and Sampson 2018]. Racket's developers provide pedagogical subsets of the language, and the Racket Stepper supports the beginner and intermediate subsets. Racket's growing language subsets partly inspired our case studies. The Racket Stepper, while useful, does not match existing visualizations nor generalize to other languages. LambdaLab is a web-based interactive tool for visualizing the execution of lambda calculus with macros and different evaluation strategies. LambdaLab's continuity is stronger than Theia's, because it shows beta reduction explicitly by connecting function arguments to their substitution sites; however, LambdaLab is not generic and only supports constructs like let bindings and data types using macros.

7 Discussion

The insight that abstract machines are formalized notional machines enabled Theia's development, but its impact is not limited to visualization. The key advantage of using a formal AM definition is that it is complete and machine-readable.

The completeness of formal machine descriptions allows for more precise comparisons of languages both between and within programming paradigms. Existing NM descriptions are imprecise and ill-defined (see Section 2.2). This is acceptable when doing informal comparison across language paradigms, but makes intra-paradigm comparison difficult. Complete descriptions make fine-grained language comparisons possible and rigorous when compared the informal analyses performed today. Completeness also allows one to empirically evaluate PSVs. For example, though Levy et al. introduce completeness and continuity for their design rationale, the authors do not explicitly evaluate Jeliot on these principles. We speculate this is because CCC cannot be formally stated without a complete characterization of a NM. In contrast, an AM description allows one to reason precisely about the state and transitions that must be present to have a complete and continuous visualization. It is possible one could formally phrase these properties and then formally *prove* that an AM-based visualization satisfies CCC.

Machine-readability is also important, because it enables generic, automated tools to manipulate and reason about NMs. For example, it enabled Theia to automatically generate visualizations. PLTutor offers another example. It uses path coverage of an interpreter to produce a complete set of educational examples. This approach could be generalized to an interpreter for an arbitrary AM.

Furthermore, if students learn *some* precise, but possibly incorrect, model of a language's semantics, it should be possible to infer this model given a student's answers to questions and handwritten traces of example programs. If one uses formal language descriptions, this inference could be automated, perhaps using insights from Saarinen et al. to generate informative questions and answers and from Feldman et al. to infer student misconceptions.

Thus complete descriptions allow one to reason precisely about language semantics, and machine-readable descriptions allow for new, more flexible educational tools.

8 Future Work and Conclusion

What's next for Theia? Though we believe abstract machines can provide a *lingua franca* for PL and CER researchers, to the best of our knowledge no formal semantics accurately captures the way most students, and even programmers in general, visualize and reason about program execution mentally or manually. Most abstract machines are designed for semantic clarity when performing proofs or specifying a language, but not for visualizations or education. For Theia to gain adoption in classrooms, one must construct AMs that capture existing mental models.

While visualizing program state completely is useful for understanding the low-level details of a language semantics, reasoning about more complex programs requires visualizing abstractions of program state that highlight or summarize relevant information and hide distracting details.

In order to scale, Theia needs to support not only generic AM frameworks like K, but also language-specific debuggers to provide PSVs of implementation-level semantics. Additionally, though Theia provides better continuity than many existing PSVs, its IR does not encode state transitions. This information would enable higher-fidelity transitions that use animations and color similar to those found in tools such as Sainati and Sampson and Woos et al.

By combining insights from PL and CER, Theia should enable instructors to more easily prototype PSVs spanning a larger range of language constructs than previous tools.

Acknowledgments

Thanks to Amy J. Ko and her students, especially Greg Nelson and Benji Xie, for providing early insightful feedback on NMs, visualization techniques, and CER. Thanks to Colleen Lewis for discussing the significance of formal semantics in the classroom. Thanks to Sorin Lerner for providing feedback on Theia's continuity and asking deep questions about the K semantics. Thanks to Dan Grossman for suggesting an effective argument structure for this paper. Thanks to Dan Grossman, Talia Ringer, Maaz Ahmad, and Sarah Chasins for reviewing early drafts of this work. Thanks to the PLSE Lab for providing feedback on an early talk, closure visualizations, and the key ideas of this paper. Finally, thanks to the anonymous referees for their informative feedback.

References

Dave Berry. 1990. *Generating program animators from programming language semantics.* Ph.D. Dissertation. University of Edinburgh.

M. Berry and M. Kölling. 2016. Novis: A Notional Machine Implementation for Teaching Introductory Programming. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 54–59. https://doi.org/10.1109/LaTiCE.2016.5

Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 445–456. https://doi.org/10.1145/2676726.2676982

Olivier Danvy and Andrzej Filinski. 1989. A Functional Abstraction of Typed Contexts.

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, 1133–1148. https://doi.org/10.1145/3314221.3314601

Stephan Diehl and Thomas Kunze. 2000. Visualizing principles of abstract machines by generating interactive animations. *Future Generation Computer Systems* 16, 7 (2000), 831–839.

Toby Dragon and Paul E. Dickson. 2016. Memory Diagrams: A Consistant Approach Across Concepts and Languages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 546–551. https://doi.org/10.1145/2839509.2844607

Molly Q. Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. 2018. Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 264, 12 pages. https://doi.org/10.1145/3173574.3173838

Matthias Felleisen and D Friedman. 1986. Control Operators, the SECD Machine, and the λ-Calculus, Formal Description of Programming Concepts III (ed. M. Wirsing), 193–217.

Robert Bruce Findler. 2014. DrRacket: The Racket Programming Environment. (2014).

Paul Gries and David Gries. 2002. Frames and folders: a teachable memory model for Java. *Journal of Computing Sciences in Colleges - JCSC* (01 2002).

Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2103656.2103691

Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally Fincher and Anthony Robins (Eds.). Cambridge University Press, Cambridge, Chapter 13, 377–413.

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. 2003. The Jeliot 2000 Program Animation System. *Comput. Educ.* 40, 1 (Jan. 2003), 1–15. https://doi.org/10.1016/S0360-1315(02)00076-3

Colleen M. Lewis. 2014. Exploring Variation in Students' Correct Traces of Linear Recursion. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. ACM, New York, NY, USA, 67–74. https://doi.org/10.1145/2632320.2632355

Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 373–376. https://doi.org/10.1145/989863.989928

Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. 2007. WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization. *SIGPLAN Not.* 42, 3 (March 2007), 14–23. https://doi.org/10.1145/1273039.1273042

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 346–356. https://doi.org/10.1145/2737924.2737991

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Sam Saarinen, Shriram Krishnamurthi, Kathi Fisler, and Preston Tunnell Wilson. 2019. Harnessing the Wisdom of the Classes: Classsourcing and Machine Learning for Assessment Instrument Generation. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 606–612. https://doi.org/10.1145/3287324.3287504

Daniel Sainati and Adrian Sampson. 2018. LambdaLab: an interactive λ-calculus reducer for learning. 10–19. https://doi.org/10.1145/3310089.3313180

Jorma Sajaniemi and Marja Kuittinen. 2008. From Procedures To Objects: A Research Agenda For The Psychology Of Object-oriented Programming Education. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* 4 (05 2008). https://doi.org/10.17011/ht/urn.200804151354

Carsten Schulte and Jens Bennedsen. 2006. What Do Teachers Teach in Introductory Programming?. In *Proceedings of the Second International Workshop on Computing Education Research (ICER '06)*. ACM, New York, NY, USA, 17–28. https://doi.org/10.1145/1151588.1151593

Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13 (06 2013), 8:1–8:31. https://doi.org/10.1145/2483710.2483713

John Whitington and Tom Ridge. 2017. Visualizing the evaluation of functional programs for debugging. (2017).

Doug Woos, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2018. A Graphical Interactive Debugger for Distributed Systems. *CoRR* abs/1806.05300 (2018). arXiv:1806.05300 http://arxiv.org/abs/1806.05300