Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices

Paul E. Dickson
pdickson@ithaca.edu
Ithaca College
Ithaca, NY, USA

Neil C. C. Brown
neil.c.c.brown@kcl.ac.uk
King's College London
London, England

Brett A. Becker
brett.becker@ucd.ie
University College Dublin
Dublin, Ireland

ABSTRACT

The term "the machine" is commonly used to refer to the complicated physical hardware running similarly complex software that ultimately executes programs. The idea that programmers write programs for a *notional machine*—an abstract model of an execution environment—not the machine itself, has risen to the point of gaining acceptance as a useful device in computing education. This has seeded a growing discussion about how explicitly utilizing notional machines in teaching can help students construct more accurate mental models, which is essential for learning programming.

Much of the existing literature necessarily involves specific languages, visualization, and/or facilitating tools, and is not very accessible to many practitioners. Less focus has been put on how teachers can make explicit use of notional machines in their teaching. In this paper we describe notional machines and their use in a manner that is more accessible to a general educator audience in order to facilitate more effective computing education at all levels. We advocate explicitly delineating between visualization tools and the notional machines they depict, isolating and clarifying the notional machine so that it is conspicuous, apparent and useful. We present examples of how this approach can facilitate a more consistent method of teaching computing, and be used in more effective pedagogical practice for teaching computing.

CCS CONCEPTS

• **Social and professional topics → Computer science education**; *CS1*.

KEYWORDS

code tracing; code writing; memory diagrams; notional machines; pedagogy; program construction; stack traces; visualization

ACM Reference Format:
Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–19, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341525.3387404

1 INTRODUCTION

The concept of a *notional machine* was introduced by du Boulay et al. [13] and further explored by du Boulay [12] in the 1980s [29]. In the late 1990s it was advocated in Ben-Ari's work on constructivism in computing education [2] but did not gain significant momentum until around a decade later. In 2012 Sorva completed a dissertation focused on the concept [29] closely followed by related work including [30]. Since then interest has grown; the ACM Digital Library shows only 27 articles prior to 2012, but 89 since. In 2019 there was a Schloss Dagstuhl seminar on the topic [15] and in 2020 a planned ITiCSE working group. Nonetheless, the recently published Cambridge Handbook of Computing Education Research notes: "Despite some noticeable treatment in the literature, notional machines do not feature prominently in curricula or texts for computing courses" [20, p383], and a recent comprehensive review of teaching introductory programming [22] notes that notional machines are occasionally used for teaching but cites only one example [5]. Despite this, topics related to notional machines such as conceptual and cognitive issues, visualization, reading/writing/-tracing, and misconceptions have received considerable attention in recent years [1].

At the highest level, the concept is that programmers do not construct programs to be executed on an actual "concrete" computer about which they know all necessary detail, but instead construct programs to be executed on an abstraction of the concrete computer: a so-called notional machine. For example, a notional machine for C might describe the concept of variables, but make no mention of whether variables are stored in memory or registers, which is a hardware detail abstracted away by that notional machine. In the words of du Boulay et al.: "A notional machine is the idealized model of the computer implied by the constructs of the programming language" [13, p237]. These abstractions are by definition consistent with the behavior of the concrete machine they represent in the current situation, including detail necessary for program construction, but omitting much or all unnecessary detail. Regarding consistency, Berry & Kölling stated: "Whatever the preferred abstraction level, it is important that the notional machine is complete and consistent: it must be able to explain all observable behaviour of the real machine, and reasoning about the notional machine must allow accurate predictions to be made about behaviour of the real machine" [4, p22]. However, this must be balanced with simplicity, as novices begin programming with very little idea of the properties of the notional machine implied by the language they are learning [13]. Nonetheless, they are models, and models have their bounds. However if these bounds are carefully aligned with the context of use, consistency should be achievable.

For novices with no effective model to start with, being presented with one could be beneficial, as explained by Ben-Ari [2, p260]:

> A (beginning) CS student has no effective model of a computer. The computer forms an accessible ontological reality. By effective model, I mean a cognitive structure, that the student can use to make viable, constructions of knowledge, based upon sensory experiences, such as reading, listening, to lectures and working with, a computer. By accessible ontological reality, I mean that a 'correct' answer is easily accessible, and moreover, successful performance requires that a normative model of this reality must be constructed.

Notional machines do not necessarily need any special notation; they just need to convey descriptions of semantic behavior. Educators may already use something similar, without explicitly calling it a notional machine, or knowing that it may indeed be a form of one. The advantage of explicitly acknowledging and pointing this out is that we can discuss and compare notional machines more easily, much as naming programming patterns (e.g. the visitor pattern) is useful for discussing programming at a higher level.

Figure 1 shows the relationship between the physical computer, its notional machine, and the student/programmer with their mental model. When we teach computer science we are often not attempting to make the student's mental model of how the *physical computer* behaves as accurate as possible. Instead we are attempting to make the student's mental model of the *notional machine* as accurate as possible. We argue that explicitly recognizing this model—and leveraging the simplicity and consistency that it can afford—may lead to more effective teaching for general and introductory-level programming, including at pre-university levels.

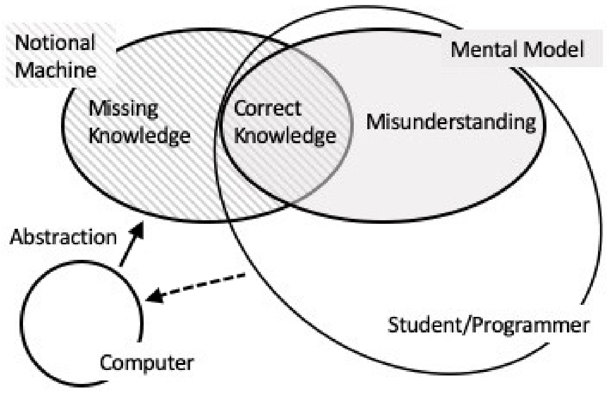

Figure 1: Relationship between the student, their mental model, a notional machine and a physical computer.

In order to grasp the notional machine concept, why it is important, and how to leverage it, an understanding of several relationships is required. Figure 1 depicts the relationships between the computer, notional machine, mental model, and the programmer. The dashed arrow represents the programmer explicitly and purposely writing code to act on the physical computer. When the student writes code they are not thinking of the execution in terms of electrons, transistors or even machine instructions. Instead when they program they usually have a mental model of a notional machine. The notional machine is a consistent abstraction of how the physical computer works (at least, consistent to a degree necessary to complete the task at hand). For example, a notional machine for Java or Python may include the concepts of references, but make no mention of memory addresses, which are considered an implementation detail unnecessary in notional machines for those languages. As mentioned above, a notional machine is a consistent abstraction of an execution environment. However this is distinct from the programmer's mental model which, unlike the notional machine, may be inconsistent or incorrect (in the given context). Conceptually, the more proficient a programmer is, the greater the overlap (correct knowledge in Figure 1) between the notional machine and that programmer's mental model. Ideally, the mental model and notional machine completely overlap when a student has become expert with that notional machine, in that context.

Computer programs typically have an abundance of state but by default, only explicit program output is visible to the programmer. There is a long history of tools for visualizing the internal memory state while a program runs, which can have a positive impact while teaching [23]. While these tools have always had an implicit notional machine as their basis for representation, it is only recently that some focus has shifted to explicitly describing notional machines and how the visualization supports them, makes them easier to comprehend, or more useful [6, 8, 17]. In this paper we continue this separation of visualizations from notional machines.

The purpose of this paper is to help make the use of notional machines accessible to a general educator audience. We begin in Section 2 with a discussion of work on program visualization and notional machines, before describing some key points about notional machines and their relation to visualizations in Section 3. Section 4 follows with examples of these relations and differences, to help elucidate the concept for those not deeply immersed in the research literature. In Section 5 we discuss the use of notional machines in teaching, before offering our conclusions in Section 6.

2 RELATED WORK

Before discussing the relationship between notional machines and the memory visualization tools associated with them, we briefly examine the literature on each.

2.1 Program Memory Visualization

Understanding what information computers store, and how it is stored, is fundamental to programming. Sorva covers these topics in detail, including the direct link between the ability to program and the ability to draw or otherwise visualize what is occurring in the computer memory [29, 31]. These visualizations are often referred to as stack traces or memory diagrams. These can be drawn by hand [7, 11, 16, 23] or generated automatically by software [9, 10, 31, 33].

While details of these visualization methods vary, what they have in common is the representation of the state of computer memory during program execution. Different methods of displaying data structures may be employed to present memory information in what the designers believe to be the most intuitive format. Different methods also display varying amounts of history of prior state with automatically generated diagrams typically replacing values associated with variables as a program is stepped through, while hand drawn diagrams usually cross-out old values before writing in new ones, leaving a history of prior state. Wilson et al. go so far as to trace each line of code almost as one would do when showing every substitution and step in solving a mathematical equation [35].

An interesting offshoot of visualization that may (depending on circumstances) more directly connect with notional machines is studying student visualizations to better understand where gaps in their knowledge exist. There has been preliminary work conducted in this area. For instance, Cunningham et al. analyzed student traces to understand where their mental models fail to match a notional machine specification [6, 7].

We make no judgment as to what method of visualization is best and wish only to point out the line delineating visualizations and notional machines. Examples later in the paper use Ithaca Memory Diagrams (IMDs) [11] which are a fairly well-defined standard, and a whiteboard-based visualization.

2.2 Notional Machine Literature

The work of Sorva et al. [28, 30] and Berry & Kölling [3, 4] has been amongst the most focused on defining notional machines recently. Berry & Kölling stated that "The notional machine offers an abstraction of the physical machine designed for comprehension and learning purposes." [3, p25], and also mention the distinction between notional machines and visualizations. Sorva had previously stated that "The student uses a given visualization of a so-called notional machine, an abstract computer, to illustrate what happens in memory as the computer processes the program." [29]. This states clearly that a visualization and a notional machine are distinct, yet related, entities. Similarly, Berry & Kölling [4, p21] stated: "We introduce the notional machine & a graphical notation for its representation." The key to both of these statements is that like a concrete computer, a notional machine can be represented by a visualization, and that they are not necessarily the same thing. Approaching notional machines from a different direction, when someone traces the execution of a piece of code they are running their mental model of a notional machine [30]. This reinforces that a notional machine is an accurate abstraction of the computer and different from the mental model, in that the mental model may include misunderstanding. Running the mental model of a notional machine in order to trace code means applying a set of rules as to what code does to change the state of the notional machine.

The concept of a notional machine is obviously intricately related to that of a semantics in programming language terminology—a relationship explored in a recent Schloss Dagstuhl seminar. The resulting report [15, p2] stated "Every semantics has an intended audience. Formal semantics typically assume a readership with high computing or mathematical sophistication. These therefore make them inappropriate for students new to computing. What forms of description of behavior would be useful to them? In computing education, the term notional machine is often used to refer to a behavior description that is accessible to beginners."

Although introduced in the 1980s by DuBoulay et al. [12, 13] it wasn't until around 2008 that work on notional machines started to appear in the literature with focus, for instance [8, 28]. The work of Sorva was influential in this period [29–31] followed by other authors through 2019, including [3, 4, 6, 7, 10, 14, 17, 21, 26, 34, 35].

Some of this work is more focused on memory visualization [8, 10, 21, 24, 34], where notional machines are more secondary. Part of the reason we are attempting to draw a line between the visualization of memory and the notional machine concept is that we believe it is valuable for teachers to be able to choose the best visualization for their particular context to present with a notional machine. Research into different methods of visualizing memory, including their pros and cons, is therefore extremely valuable but in this case not relevant to the argument we are trying to make.

Johnson et al. made observations supporting the case for notional machines, stating: "Our results indicate that teaching Python alone (i.e., without reference to an explicit notional machine) leads to the formation of misconceptions and misunderstandings by students" [18, p1]. They claim that Python's simple structure hides a deceptively complex notional machine, and that teaching Python alongside a simpler language (simple enough to reveal a notional machine similar to that of Python) aids the development of a more robust mental model.

Others have described using notional machines as part of an approach to reaching different goals. Hidalgo et al. [17] and Sorva & Seppälä [32] use notional machines as a basis for rethinking how to teach CS1. Fisler et al. [14] discuss notional machines in relation to knowledge transfer in one of the few papers that is not focused on CS1. Out of the same group came another paper focused on recursion that includes visualization of the actual mathematical steps the code runs through while processing, along with memory state [35]. Cunningham et al. have done interesting work showing how analysis of student generated visualizations can show problems with student models of notional machines [6, 7].

3 NOTIONAL MACHINES & VISUALIZATION

Some of the existing work on visualizing notional machines can be confusing, as many papers refer to "the notional machine" and then discuss a particular visualization used. There are several key tenets to understand, which we hope help illuminate and clarify the difference between the two:

- Any given programming language can have a wide variety of notional machines, which may be at different levels of complexity.
- The suitability (and complexity) of a notional machine depends on the context of its use.
- A notional machine and a visualization usually relate closely, but a notional machine may have multiple possible visualizations, and a visualization may be useful for multiple notional machines.

We will explain each of these points in turn.

3.1 Myriads of Notional Machines

Programming languages are rich and complex constructs as are the machines that execute them. Partially for this reason there is no single notional machine for a given programming language. Consider for example, evaluating a simple expression such as $(1+2)\times 5$. One notional machine could refer to using a stack for evaluation, transforming the expression to reverse Polish notation and pushing the result of 1 + 2 on to the stack, before popping it to multiply by 5. Another notional machine could refer to substitution, explaining that the result of 1 + 2 is substituted into the original expression to give 3×5. Both of these notional machines are correct, and both omit unnecessary detail (for example, the fact that the numbers are represented in binary in a computer is an unnecessary detail here as it often is), but they take different approaches to abstracting and modeling the execution.

As well as having notional machines with different approaches as above, notional machines often have a subset relation, especially when teaching. A simple notional machine for Java in the first week may treat `int` and `String` as simple atomic types. The details about `String` being a class and the variable being a reference to an object with an internal array of Unicode characters could be initially omitted—but a more complex notional machine that included more of these details could be explained in later teaching when useful.

3.2 Notional Machines, Suitability, & Context

This example leads into our next point. A notional machine for early teaching may be grossly simplified. This may cause some instructors to feel uneasy, as if they are teaching something that is incorrect. However, incompleteness should not be confused with incorrectness. When a child writes their first English sentence we tell them that the first word starts with a capital but other words do not. When we then add in that names also begin with a capital letter, it should be seen as adding detail, not that the earlier rule was incorrect. Both are correct and consistent—in the context of the student level, the task, and the goal of that instance.

Simple notional machines make sense for early teaching (as originally stressed by du Boulay et al. [13]) and more complex notional machines for later teaching. However, no notional machine really reaches completeness, and all (by definition) omit unnecessary detail. What detail is unnecessary is often contextual, and a matter of fine judgment, for both educators and professionals. For example, many programmers in high-level languages do not concern themselves with the fine detail of which memory is likely to be cached, and just ignore memory access speed in their notional machine, whereas others who are in a context where performance is of great importance may include that as part of their notional machine.

3.3 Notional Machines & Visualizations

A visualization is invariably tied to particular features of a notional machine. For example, a visualization depicting a stack during an expression's evaluation is implicitly based on a notional machine with stack evaluation. A visualization typically shows some changes in state, and thus it relates to a notional machine that features that state. To help elucidate this concept, the next section features examples of visualizations and notional machines, and how they relate to each other.

4 EXAMPLES

In this section, we present two concrete examples. In the first we show how a single notional machine can be used with different visualizations, and how different visualizations can highlight different features and connotations. In the second example, we show how one visualization may relate to different notional machines.

4.1 Basic Variable Example with Multiple Visualizations

Consider the following description of a very basic notional machine used in many first programming classes:

> Variables are names that have values assigned to them. An assignment statement stores the result of the right-hand side of the expression into the variable on the left-hand side, replacing any previous value.

The code sample in Listing 1 shows a small piece of code that initializes two variables then re-assigns one, to help demonstrate this notional machine. We will describe two different visualizations of executing this code in this notional machine.

Listing 1: Simple variable example code.

```
int num1 = 4
int num2 = 5
num1 = num1 + num2
```

4.1.1 Memory diagram. When the code from Listing 1 is traced using an Ithaca Memory Diagram (IMD), the result is shown in Figure 2. The variables created in memory are shown on the left while the values that are stored for them on the stack are shown on the right. The memory diagram shows state as it is updated as each line of code is executed. When the first line of code runs, the `num1` and the `4` are written into the diagram. The second line of code results in `num2` next to `5`, and the third line results in the `4` being crossed out and `9` written. The IMD shows previous values stored in memory by virtue of them still appearing on the stack though now crossed out to show that the value no longer exists.

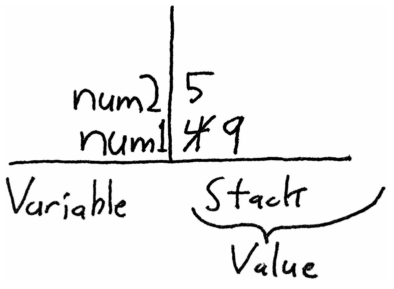

Figure 2: IMD trace of Listing 1.

4.1.2 Whiteboard associations. Figure 3 shows an alternative visualization where variables are listed on the left, values on the right, and where lines between the two are redrawn when values change. So after the first two lines (of code), *num1* has a line to *4*, and *num2* has a line to *5*, but after the third line (of code), the line from *num1* is erased, and a new line drawn from *num1* to *9*.

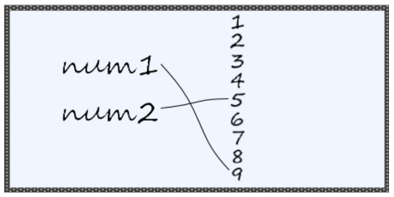

Figure 3: Trace of Listing 1 simulated on a whiteboard.

4.1.3 Notional machine. Both of these visualizations are a consistent representation of the single notional machine explained earlier. However, they have different features, which can have pros and cons. For example, the IMD shows the history of the variables, whereas the whiteboard does not. The IMD thus allows students to see the history of the execution, but also invites a possible misconception that the variable may somehow remember its previous values, or store more than one at a time. Alternatively, the whiteboard example persistently shows the numbers 1-9, which may give rise to the misconception that only these numbers are representable in variables. Our key point is that although these visualizations faithfully depict the same notional machine, they have different affordances and connotations.

4.2 Array Example With Multiple Notional Machines

The second example (Listing 2) has a C-like syntax for assigning arrays. We will give two different notional machines that could be used to execute this code:

- The first has immutable arrays. When an array is created, it is allocated on the heap, but its values can no longer be changed. Arrays are garbage-collected when no longer in use.
- The second has mutable non-shared arrays that has storage allocated on the heap, but they are deallocated immediately when the reference is overwritten or goes out of scope.

Listing 2: Basic array example code.

```
int[] array1 = [1, 2, 3]
array1 = [4, 5, 6]
```

Figure 4 shows a visualization of Listing 2 generated by an IMD trace of the code. The first line results in the variable `array1` with a memory address of `0xA`[1] on the stack and values stored on the heap with a location of `0xA`. An arrow and memory addresses are used to show that the memory address on the stack corresponds to the location on the heap. The second line results in a new location (`0xB`) and values on the heap and a replacement of the location stored on the stack, with the old location and arrow to the location on the heap crossed out. A red line is used to cross out the data at memory location `0xA` to indicate that the memory is deallocated.

[1]In IMDs, memory locations are randomly assigned and come in the form of 0x followed by a hexadecimal number. This is done to help illustrate that the data is stored on the heap in a location and that the location is stored on the stack associated with the variable but without having to worry about how the heap address is assigned. The data is also shown as a contiguous chunk of memory for clarity even if values may not be contiguous on the heap.

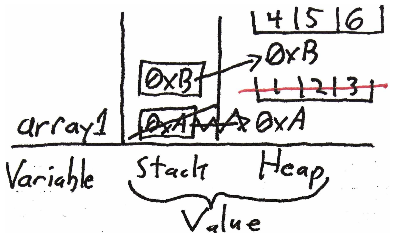

Figure 4: IMD trace of Listing 2. The line through the values in address *0xA* indicate that they have been deallocated.

The mechanism for indicating that memory is released on the heap in an IMD is the line through that data.

The single IMD visualization applies equally to both of our quite different notional machines, because they share the same array creation and deallocation semantics in this example. So a visualization does not have to be unique to one notional machine, and indeed many common visualizations of variable state can apply to a large range of notional machines, either for the same language, or across different languages. For instance, basic variable manipulation can be visualized the same way for C, Java, Python and so on.

5 NOTIONAL MACHINES AS A PEDAGOGICAL TOOL

In this section we discuss the use of notional machines as an explicit concept that instructors can utilize in practice. This was advocated in the late 1990s by Ben-Ari who noted [2, p260]:

> ...the model of a computer—CPU, memory, I/O peripherals—must be explicitly taught and discussed, not left to haphazard construction and not glossed over with facile analogies. Teaching the model can be done using *epistemic games*—formalized procedures for constructing knowledge—such as a model computer [27] or a 'notional machine' [12].

5.1 Notional Machine Considerations

Notional machines provide advantages and disadvantages. They can be used to make apparent to students that the notional machine is a model of program execution, but not perhaps the ground truth. This is consistent with the fact that much learning is often accomplished with models. Disciplines such as physics (e.g. Newton vs relativity) and chemistry (e.g. models of atoms) have a history of explaining simple models to beginners that are expanded in later teaching. By explicitly conveying that our execution models are indeed models, we set the stage for properly scaffolding later learning.

Like choosing a programming language [22] or environment [19], it can be difficult to choose a good notional machine. Apart from being infinite in number, many instructors have learned modern languages like Java by adapting their knowledge of languages like C. Consider the broad-brush statement that Java is like C but with objects instead of structures. Similarly, instructors may try to explain a notional machine for Java with memory addresses for references, replicating their own learning journey [25]. However, Java permits no pointer manipulation, so a simpler and more or less equivalent notional machine can just omit the idea of memory addresses in favor of the heap as an abstract set of objects. However, care must be taken when doing so. Similarly, instructors can often be tempted to include too much detail. In the course of conducting teacher training, the authors have often heard statements like "of course, the students must understand assembly". The temptation to reach to the next layer down is exactly what notional machines should help to prevent. If the machine is consistent, it is unnecessary to teach a lower level. Trying to teach the entire implementation of a computer in the first course stands to erode the gains brought about by higher level abstractions and higher level languages which have advanced programming practice.

5.2 Use of Visualizations

The earlier examples illustrate how a memory visualization may be considered separately from the notional machine as a whole, and how visualizations can be chosen semi-independently of the notional machine. This means that there remains freedom in the choice of visualization tool to use for a given course, even once a notional machine is chosen. It also means that courses that are currently built around memory visualization tools can add in a notional machine component without requiring a shift of visualization method. Finally, it also suggests that it may be possible to use the same visualization method between multiple courses with a basis in notional machines, in an effort to reduce extraneous cognitive load. As the notional machine used in a particular class need only be consistent for the material presented at the time it may be possible to start with a simplified version of a visualization method that gradually adds detail as concepts become more complex in a course or between courses, ensuring that the particular notional machine always meets the requirement of being consistent. This is akin to language levels—a feature offered by languages such as Racket.[2]

Another consideration is that having a clearer picture of what a notional machine is (versus a memory visualization or a mental model) can help clarify concepts for students. At a minimum, having educators with a clearer view of their mental model / visualization / notional machine relationship must have positive effects. By using multiple memory visualization tools in combination with the same set of rules for how code corresponds to computer memory, we can show the same concepts from multiple perspectives. This can lead to discussions with students of how tracing code is in fact them running their mental model of a notional machine. It helps students to understand why it is important that they can create memory visualizations from code. Seeing where their diagrams are wrong exemplifies where their mental models are inaccurate in relation to the notional machine—which in most cases is what educators teach students to program, whether intentionally or not.

5.3 Teaching Strategies

The challenge in a constructivist notional machine framework is having students construct a correct (or at least functional) mental model from the presented notional machine. It seems a reasonable approach to explicitly teach that notional machine, which can be conceived of as the ideal mental model we would like students to have (for present requirements). In other words if teachers explicitly teach a notional machine, chances of success could be greater.

[2]https://racket-lang.org/

One way of explicitly referencing the use of a notional machine in class would be in relation to teaching pointers in C/C++. In this situation IMDs explicitly list a memory address for anything on the stack as a two-digit hexadecimal number and anything on the heap as a 3-digit hexadecimal number. In actuality all of the memory addresses will be of the same format and longer than 2-3 digits (often demonstrated by printing actual memory addresses in code examples)—but by utilizing this abstraction in our notional machine we are drawing student attention to the differences in storage location. We have used this in class and found that students understanding of pointers seemed to be helped by an explicit reference to a notional machine and why the notional machine abstracted the way it did in the example.

Imagine a textbook for introductory programming that is built around presenting a notional machine and building it from one concept to the next. In the chapter where variables are introduced the notional machine from Section 4.1.3 could be included, for instance. With each new concept, depth can be added to the notional machine being presented in the text. The additional information would not change any of the results presented previously but would instead just add detail as required to obtain new results from more complex problems. Such an incremental development (especially if students are a witness to, or participant in the shaping of this evolving notional machine) may avoid misconceptions that would otherwise be encountered due to students trying to fill in flaws or gaps in their learning. We believe that this what some textbooks already do, but most often rather implicitly. By drawing out the explicit concept of the notional machine, both authors and readers can be clearer that the rules of the notional machine are being expanded and built upon—abstraction in action.

6 CONCLUSIONS

Notional machines, code tracing, mental models and program visualization are often appear conflated in the literature at least to non-experts. In this position paper we sought to clarify the differences between these, presenting notional machines in a manner that is hopefully more accessible to a general audience.

We demonstrated that from a teaching point-of-view, program visualization and notional machines are two different things: the former is about presenting program state, and the latter is about the rules that govern how that state comes about. With these two concepts decoupled, it can enable us to teach each more precisely. With regard to visualizing memory, it suggests that a generic visualization method that can work for multiple languages would be most effective in teaching as it would allow the same visualization to be used for multiple notional machines. This should make it easier for courses to just focus on the differences in the rules that govern the notional machine and therefore reduce cognitive load.

We believe that explicitly discussing notional machines will help educators be more clear about their own practice, in a similar way to programmers discussing concepts like code smells or decoupling can allow a clearer discussion of program construction. By explicitly and intentionally separating the notional machine from adjacent concepts like mental models and visualization, we can deliberately design and use appropriate notional machines at each stage in our teaching, and achieve clearer understanding among our students.

REFERENCES

[1] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432

[2] Mordechai Ben-Ari. 1998. Constructivism in Computer Science Education. In *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98)*. Association for Computing Machinery, New York, NY, USA, 257–261. https://doi.org/10.1145/273133.274308

[3] Michael Berry and Michael Kölling. 2013. The Design and Implementation of a Notional Machine for Teaching Introductory Programming. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSE '13)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/2532748.2532765

[4] Michael Berry and Michael Kölling. 2014. The State of Play: A Notional Machine for Learning Programming. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 21–26. https://doi.org/10.1145/2591708.2591721

[5] Michael Berry and Michael 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.

[6] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 164–172. https://doi.org/10.1145/3105726.3106190

[7] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice Rationales for Sketching and Tracing, and How They Try to Avoid It. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 37–43. https://doi.org/10.1145/3304221.3319788

[8] Tonci Dadic, Slavomir Stankov, and Marko Rosic. 2008. Meaningful Learning in the Tutoring System for Programming. In *ITI 2008-30th International Conference on Information Technology Interfaces*. IEEE, 483–488.

[9] Andrew R. Dalton and William Kreahling. 2010. Automated Construction of Memory Diagrams for Program Comprehension. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*. ACM, New York, NY, USA, Article 22, 6 pages. https://doi.org/10.1145/1900008.1900040

[10] Peter Donaldson and Quintin Cutts. 2018. Flexible Low-cost Activities to Develop Novice Code Comprehension Skills in Schools. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education (WiPSCE '18)*. ACM, New York, NY, USA, Article 19, 4 pages. https://doi.org/10.1145/3265757.3265776

[11] 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

[12] Benedict du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[13] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies* 14, 3 (1981), 237–249.

[14] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 213–218. https://doi.org/10.1145/3017680.3017777

[15] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019), 1–23. https://doi.org/10.4230/DagRep.9.7.1

[16] Matthew Hertz and Maria Jump. 2013. Trace-based Teaching in Early Programming Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[17] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, and Vladimir Lara-Villagrán. 2016. Understanding Notional Machines Through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing. *CLEI Electronic Journal* 19, 2 (2016), 3–3.

[18] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of Student Misconceptions Using Python as an Introductory Programming Language. In *Proceedings of the 4th Conference on Computing Education Practice 2020 (CEP 2020)*. Association for Computing Machinery, New York, NY, USA, Article Article 4, 4 pages. https://doi.org/10.1145/3372356.3372360

[19] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 759–765. https://doi.org/10.1145/3328778.3366882

[20] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Chapter 13, 377–413. https://doi.org/10.1017/9781108654555.014

[21] Derrell Lipman. 2014. LearnCS!: A New, Browser-based C Programming Environment for CS1. *J. Comput. Sci. Coll.* 29, 6 (June 2014), 144–150. http://dl.acm.org/citation.cfm?id=2602724.2602752

[22] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[23] Thomas L. Naps, Guido Rossling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Angel Velazquez-Iturbide. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.* 35, 2 (June 2002), 131–152. https://doi.org/10.1145/782941.782998

[24] Greg L. Nelson, Benjamin Xie, and Andrew 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

[25] Amanda Oleson and Matthew T. Hora. 2014. Teaching the Way They Were Taught? Revisiting the Sources of Teaching Knowledge and the Role of Prior Experience in Shaping Faculty Teaching Practices. *Higher Education* 68, 1 (01 Jul 2014), 29–45. https://doi.org/10.1007/s10734-013-9678-9

[26] Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: Automatically Generating Correct Program State Visualizations. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/3358711.3361625

[27] Lorraine Sherry. 1995. A Model Computer Simulation as an Epistemic Game. *SIGCSE Bull.* 27, 2 (June 1995), 59–64. https://doi.org/10.1145/201998.202016

[28] Juha Sorva. 2008. The Same but Different Students' Understandings of Primitive and Object Variables. In *Proceedings of the 8th International Conference on Computing Education Research (Koli '08)*. ACM, New York, NY, USA, 5–15. https://doi.org/10.1145/1595356.1595360

[29] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education; Visuaalinen Ohjelmasimulaatio Ohjelmoinnin Alkeisopetuksessa.* Ph.D. Dissertation. Aalto University, Espoo, Finland. http://urn.fi/URN:ISBN:978-952-60-4626-6

[30] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[31] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.* 13, 4, Article 15 (Nov. 2013), 64 pages. https://doi.org/10.1145/2490822

[32] Juha Sorva and Otto Seppälä. 2014. Research-based Design of the First Weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2674683.2674690

[33] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/1930464.1930471

[34] Li Sui, Jens Dietrich, Eva Heinrich, and Manfred Meyer. 2016. A Web-Based Environment for Introductory Programming Based on a Bi-Directional Layered Notional Machine. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. ACM, New York, NY, USA, 364–364. https://doi.org/10.1145/2899415.2925487

[35] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 1023–1028. https://doi.org/10.1145/3159450.3159479