

Notional Machines and Programming Language Semantics in Education

Edited by

Mark Guzdial¹, Shriram Krishnamurthi², Juha Sorva³, and Jan Vahrenhold⁴

1 University of Michigan – Ann Arbor, US, mjguz@umich.edu

2 Brown University – Providence, US, sk@cs.brown.edu

3 Aalto University, FI, juha.sorva@iki.fi

4 Universität Münster, DE, jan.vahrenhold@uni-muenster.de

Abstract

A formal semantics of a language serves many purposes. It can help debug the language’s design, be used to prove type soundness, and guide optimizers to confirm that their work is correctness-preserving. Formal semantics are evaluated by several criteria: full abstraction, adequacy, soundness and completeness, faithfulness to an underlying implementation, and so on.

Unfortunately, we know relatively little about how non-experts, such as students, actually employ a semantics. Which models are they able to grasp? How useful are these as they explain or debug programs? How does their use of models evolve with the kinds of programs they write? And does studying these kinds of questions yield any new insights into forms of semantics?

This Dagstuhl Seminar intended to bridge this gap. It brought together representatives of the two communities—who usually travel in non-intersecting circles—to enable mutual understanding and cross-pollination. The Programming Languages community uses mathematics and focuses on formal results; the Computing Education Research community uses social science methods and focuses on the impact on humans. Neither is superior: both are needed to arrive at a comprehensive solution to creating tools for learning.

Seminar July 7–12, 2019 – <http://www.dagstuhl.de/19281>

2012 ACM Subject Classification Social and professional topics → Computing education, Theory of computation → Program semantics

Keywords and phrases computing education research, formal semantics, misconceptions, notional machines

Digital Object Identifier 10.4230/DagRep.9.7.1

Edited in cooperation with Philipp Kather



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Notional Machines and Programming Language Semantics in Education, *Dagstuhl Reports*, Vol. 9, Issue 7, pp. 1–23

Editors: Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold



DAGSTUHL Dagstuhl Reports

REPORTS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


1 Summary

Mark Guzdial

Shriram Krishnamurthi

Juha Sorva

Jan Vahrenhold

License  Creative Commons BY 3.0 Unported license
 © Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold

A formal semantics is often intended as a tool to comprehend the behavior of a language or other system. Semanticists assume, for instance, that programmers can use a semantics to understand how a particular program will behave without being forced to resort to deconstructing the output from a black-box evaluator. Indeed, different semantic models vary in what aspects of program behavior they highlight and suppress.

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.

Our meeting therefore focused on what we know, and what we need to learn, about notional machines. In particular, we studied and discussed:

- Different formulations of notional machines for a variety of languages.
- The distinction between a general description of behavior, independent of a specific program, and the explication of behavior of a specific program. We argued for the value of having both the general and the specific, since learners might need to shift between the two.
- The different forms that a notional machine can take, and their styles: [MARK fill in]
- The many analogies employed in notional machines, with their respective strengths and weaknesses.
- The different forms of theories that apply to generating and understanding notional machines, including cognitive and social.
- Analogies to notional machines in other domains, from models in physics to rulebooks in board games.

We accomplished most of our stated goals: to bring together the semantics and education communities (though with much greater representation from the latter than the former); to create tutorials to educate each on the knowledge and methods of the other; and to formulate interesting examples. While there did not appear to be many long-standing “open questions”, and there was not enough time to engage in editing Wikipedia, groups did organized community-wide activities (such as surveys to be conducted at upcoming conferences) and large banks of research questions (which are concrete and valuable outcomes that we had not anticipated). In sum, we believe the seminar successfully accomplished its overall stated goals.

2 Table of Contents

Summary

Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold 2

Overview of Talks

Drawings of Notional Machines from Secondary School Teachers
Brett A. Becker 6

Sketching Notional Machines with Meaning
Kathryn Cunningham 6

Using the Structure Behavior Function Framework to Understand Learning of
Computer Programming
Kathryn Cunningham and Mark Guzdial 7

Notional Machines and Research from the 1970s and 1980s
Benedict du Boulay 8

Runestone Interactive Ebooks with Adaptive Parsons Problems
Barbara Ericson 8

Presenting Name/Value Mappings in Notional Machines
Kathi Fisler 8

Empirical Studies
Robert L. Goldstone 9

Making Programming Languages to Meet a Greater Need
Mark Guzdial 9

Conceptual Change in Learning to Program
Matthias Hauswirth 10

What Do Students “See” in Computing Contexts?
Geoffrey L. Herman 10

Reading Code Aloud
Felienne Hermans 10

Sensing and First Data
Matthew C. Jadud 11

Giving Feedback and Hints in (Haskell/Java/...) Programming Tutors Based on
Comparing Model Solutions to Student Solutions
Johan Jeuring 11

Philosophical Concept Analysis in PL or SE or CSE or ...
Antti-Juhani Kaijanaho 11

Towards Algorithm Comprehension
Philipp Kather and Jan Vahrenhold 12

Code and Cognition Lab
A. J. Ko 12

Language Levels
Shriram Krishnamurthi 12

Explicit Programming Strategies <i>Thomas D. LaToza</i>	13
Conceptual Change & Knowledge in Pieces (KiP) <i>Colleen Lewis and Matthias Hauswirth</i>	13
Concrete Notional Machines <i>Colleen Lewis</i>	13
Reference-point Errors: Slips? or Misconceptions of the Notional Machine? <i>Craig Müller</i>	14
Notional Machines for Everyday Life <i>Andreas Mühlring</i>	14
Making a Causal Diagram for Learning Programming <i>Greg Nelson</i>	15
Pointer Concepts in C <i>Andrew Petersen</i>	15
Semantics Tutorial <i>Joseph Gibbs Politz</i>	15
Activity Theory <i>R. Benjamin Shapiro</i>	16
Stuff We Wish We Knew (About Notional Machines) <i>Juha Sorva and Otto Seppälä</i>	16
Revisiting Two Past Publications through the Lens of Notional Machines <i>J. Ángel Velázquez Iturbide</i>	17
Working groups	
Breakout Group 1 <i>Geoffrey L. Herman and Philipp Kather</i>	17
Breakout Group 2 <i>Craig Müller and Franziska Carstens</i>	18
Breakout Group 3 <i>Brett A. Becker, Neil C. C. Brown, Paul Denny, Rodrigo Duran, Robert L. Goldstone, Antti-Juhani Kaijanaho, Greg Nelson, Carsten Schulte, Otto Seppälä, and Steven A. Wolfman</i>	18
Breakout Group 4 <i>Kathi Fisler and Kathryn Cunningham</i>	19
Categorizing Notional Machines and their Representation or Visualization <i>Franziska Carstens</i>	19
Instructional Design for Notional Machines <i>Barbara Ericson, Robert L. Goldstone, Matthias Hauswirth, Antti-Juhani Kaijanaho, Greg Nelson, André L. Santos, and Anya Taftiovich</i>	20
Concept Analysis for Notional Machines <i>Antti-Juhani Kaijanaho, Thomas Ball, Markus Müller-Olm, and Juha Sorva</i>	20
Notional Machines for Everything <i>Joseph Gibbs Politz, Mark Guzdial, and Philipp Kather</i>	20

Notional Machines for Scratch and Python
Otto Seppälä, Thomas Ball, Titus Barik, Brett A. Becker, Paul Denny, Rodrigo Duran, Juha Sorva, and J. Ángel Velázquez Iturbide 21

Notional Machines and Simulation
Steven A. Wolfman, Kathryn Cunningham, Kathi Fisler, Greg Nelson, and Jan Vahrenhold 22

Participants 23

3 Overview of Talks

3.1 Drawings of Notional Machines from Secondary School Teachers


Brett A. Becker (University College Dublin, IE)

License  Creative Commons BY 3.0 Unported license
© Brett A. Becker

I presented a poster consisting of drawings of notional machines from 9 Irish Secondary School Teachers (senior cycle – teaching students 15-18 years of age). These teachers were enrolled in a 30 ECTS postgraduate diploma in Educational Studies in Computational Thinking. Specifically, these teacher-students were in my course “How Computers Work” which is a post-programming basic architecture course. At the point that the teacher-students drew their notional machines they had completed course material on: Number systems; Logic; Boolean Algebra; Von Neumann Architecture; CPU; Memory; and the Bus. They had completed tutorials on: binary / decimal conversion (by hand and in Python); creating simple logic gates in a simulator (e.g. half-adder); “anatomy of a computer” where we opened up a desktop computer; the “Little Man Computer” which is a simulator that “has many of the basic features of a modern computer that uses the Von Neumann architecture”; and benchmarking their own laptops with software. I then gave the teacher-students a 20 minute primer on notional machines. Then they drew their own depictions of what a notional machine looks like to them. There were many interesting observations made by those at Dagstuhl who viewed these depictions. It is likely that more can be learned by having students produce visualisations of their mental models and their concepts of notional machines.

3.2 Sketching Notional Machines with Meaning

Kathryn Cunningham (University of Michigan – Ann Arbor, US)

License  Creative Commons BY 3.0 Unported license
© Kathryn Cunningham

Prior research has shown that sketching out a code trace on paper is correlated with higher scores on code reading problems. Why do students sometimes choose not to draw out a code trace, or if they do, choose a different sketching technique than their instructor has demonstrated? In this study, we interviewed 13 CS1 students retrospectively about their decisions to sketch and draw on a recent programming exam. When students do sketch, we find that their sketching choices do not always align with the way that experts illustrate execution of the notional machine. Sketching choices are driven by a search for a program’s patterns, an attempt to create organizational structure among intermediate values, and the tracking of prior steps and results. When novices don’t sketch, they often report that they’ve identified the goal that the code achieves. In either case, novices are searching for the functionality of code, rather than merely tracing its behavior. Student sketches suggest new notional machine visualization approaches that integrate the meaning of code with code behavior. Title – Cognitive Complexity of Programs and Notional Machines- Rodrigo Silva Duran Instructional designers, examiners, and researchers frequently need to assess the complexity of computer programs in their work. However, there is a dearth of established methodologies for assessing the complexity of a program from a learning point of view. In

this poster, I present theories and methods for describing programs in terms of the demands they place on human cognition. More specifically, I draw on Cognitive Load Theory and the Model of Hierarchical Complexity to extend Soloway's plan-based analysis of programs and apply it at a fine level of granularity. The resulting framework of Cognitive Complexity of Computer Programs (CCCP) generates metrics for two aspects of a program: plan depth and maximal plan interactivity. Plan depth reflects the overall complexity of the cognitive schemas that are required for reasoning about the program, and maximal plan interactivity reflects the complexity of interactions between schemas that arise from program composition. To generate the aforementioned metrics, instructors need to first supply a concrete program written in a given programming language. A second input for the model is the expected prior knowledge of a given learner, measured in terms of what kinds of plans have been automated. Third, a notional machine will describe the plans and elements from the programming language required to comprehend the program, how the semantics describe the interaction among elements and to which level of detail and abstraction the plans are represented by the learner. In this poster we explore questions regarding the design of notional machines aimed to different audiences, how much detail and abstraction a notional machine should have when aimed to a particular audience and what formats could be used to communicate a notional machine more clearly.

3.3 Using the Structure Behavior Function Framework to Understand Learning of Computer Programming

Kathryn Cunningham (University of Michigan – Ann Arbor, US) and Mark Guzdial (University of Michigan – Ann Arbor, US)

License © Creative Commons BY 3.0 Unported license
© Kathryn Cunningham and Mark Guzdial

Over the past few decades, many researchers have proposed that designed devices can be understood in terms of their structure (what they are made of), their function (why they were designed), and their behavior (how they work). We unified past definitions of the Structure Behavior Function (SBF) framework, and then applied the framework to the understanding of computer programs. We defined the structure of a program to be the program's syntax and its programming plans; we defined the function of a program to be its purpose in natural language; and we defined the behavior of a program to be the way that it executes on a notional machine. In the SBF framework, the ability to transition between structure, behavior, and function is crucial to the design process. In programming education, we explicitly teach the transition from structure to behavior through tracing exercises, but the transition between behavior and function is not typically taught. We interpreted three theories of programming knowledge using the language of the SBF framework, showing that SBF can organize and relate several areas of computing education research.

3.4 Notional Machines and Research from the 1970s and 1980s

Benedict du Boulay (University of Sussex – Brighton, GB)

License  Creative Commons BY 3.0 Unported license
© Benedict du Boulay

This presentation covered four areas: six influential pieces of work of the period, difficulties of learning programming, notional machines and conclusions. Feurzeig and Papert’s work on Logo was an example of an influential piece of work. The difficulties of learning programming covered four areas: orientation, notional machines and notation, structures, and pragmatics. For each area a couple of research papers from the period were identified. Notional machines were divided into two kinds: stories and (machine generated) representations. The conclusions offered a link between notional machines and semantics via the route of formally specifying learner misconceptions.

3.5 Runestone Interactive Ebooks with Adaptive Parsons Problems

Barbara Ericson (University of Michigan – Ann Arbor, US)

License  Creative Commons BY 3.0 Unported license
© Barbara Ericson

I have been creating interactive ebooks for Advanced Placement Computer Science using principles from educational psychology: worked example plus practice, multiple modalities, and adaptive learning. Advanced Placement (AP) Computer Science courses are intended to be equivalent to college-level courses. I have been researching the effectiveness and efficiency of solving Parsons problems versus writing and fixing code. I also created two types of adaptation for Parsons problems: intra-problem and inter-problem adaptation. In intra-problem adaptation if the learner is struggling to solve the current problem it can be made easier dynamically by disabling distractors, providing indentation, or combining blocks. In inter-problem adaptation the difficulty of the next problem is adjusted based on the learner’s performance on the previous problem. The goal is to keep the learner challenged but not frustrated.

3.6 Presenting Name/Value Mappings in Notional Machines

Kathi Fisler (Brown University – Providence, US)

License  Creative Commons BY 3.0 Unported license
© Kathi Fisler

For programs without assignment statements, there are several ways to capture the mapping from names to values. The tradeoffs are particularly interesting for programs that involve data with components or objects. We illustrate the tradeoffs of two models for program evaluation for programs with such data: one substitutes the value associated with each parameter name, while the other substitutes the heap address associated with each name. A preliminary user study shows that each has advantages in some contexts, suggesting that a combination of the models be used in program tracing tools.

3.7 Empirical Studies

Robert L. Goldstone (Indiana University – Bloomington, US)

License © Creative Commons BY 3.0 Unported license
© Robert L. Goldstone

Computer programming is one of the most cognitively demanding and complex tasks in which humans engage. It places challenging demands on working memory, abstraction, mental modeling and simulation, planning, problem solving, memory retrieval, and the creation of novel, robust and flexible structures and processes. There has been an extended literature on the psychology of computer programming. Some of this has focused on the syntactic, semantic, and strategic misconceptions that students and even experts possess. Other research has described performance factors related to fragile knowledge, cognitive load, natural language intrusions, limited working memory, schema-based misconstruals, perception, transfer of knowledge, and individual differences. A theoretically and important research agenda concerns how best to enable humans to produce sophisticated computer programs that push and even transcend human physical and mental limits. Pedagogical recommendations include: integrating role-based conceptions of variables, read-trace-explain-sketch curricula, incorporating concept inventories, combining worked-out examples and test items, optimal scheduling of worked-out examples, labeling, idealization, aligning natural and formal language, explicitly training for transfer, peer instruction, and game-based components. Transcending human limitations in programming will often involve the creation of human-machine distributed cognitive systems, featuring technological innovations such as: color coding/highlighting, visual editors, notional machines, algorithm visualizations, simplified languages/environments, human-consumable error and status messages, embedded assessments, learning analytics, and creating new programming languages explicitly design to fit and shape human mental models.

3.8 Making Programming Languages to Meet a Greater Need


Mark Guzdial (University of Michigan – Ann Arbor, US)

License © Creative Commons BY 3.0 Unported license
© Mark Guzdial

There is so little computer science in high schools today, in part because it's so hard to program. I suggest that we need to figure out what makes programming more accessible. New tools like Vega-Lite and Sarah Chasin's Helena suggest a different strategy – developing task-based programming languages that serve a specific purpose and can be used successfully within 10 minutes.

3.9 Conceptual Change in Learning to Program

Matthias Hauswirth (University of Lugano, CH)

License  Creative Commons BY 3.0 Unported license
© Matthias Hauswirth

Learning to program is hard. In this poster we show two approaches we used to investigate the conceptual change students undergo as novice programmers. We use the Informa Clicker tool where students construct responses, similar to visual program simulation, and we use the Informa Mastery Learning platform to support the detailed analysis of the development of a fine-grained set of specific skills. Based on these approaches we have identified a collection of 165 misconceptions about programming in Java. At USI we are now embarking on a project to investigate trajectories through that space of conceptual understanding and to connect learning between different programming languages.

3.10 What Do Students “See” in Computing Contexts?

Geoffrey L. Herman (University of Illinois – Urbana Champaign, US)

License  Creative Commons BY 3.0 Unported license
© Geoffrey L. Herman

Cognitive research has shown that gaining expertise in a subject area both changes what an individual sees when shown a visual representation and it also changes how that individual searches for information in that visual representation. The goal of my research is to explore the connection between perception and students’ knowledge of computational notional machines. Using a mixture of eye-tracking methods and qualitative interviews, we are seeking to describe how students learn to read and trace code.

3.11 Reading Code Aloud

Felienne Hermans (Leiden University, NL)

License  Creative Commons BY 3.0 Unported license
© Felienne Hermans


When children learn to read, they almost invariably start with oral reading: reading the words and sentences out loud, not just to demonstrate their newly acquired skill, but also because they simply cannot do it in a different fashion yet. Most children take years to learn to read silently, during which they go through a number of phases including whispering and lip movement. Several studies have shown that, for novice readers, reading aloud supports comprehension. This should not come as a surprise, sometimes when reading difficult English words, I still read aloud! While we do not know exactly how reading aloud helps, the fact that it does is often attributed to the fact that reading aloud focuses your attention to the text, and thus makes it less likely that you will skip letters or words.

This made us wonder, why do we not practice to read code aloud? In the same way that reading text aloud helps to understand meaning, so could reading source code! We call this idea code phonology. Settling on a phonology could be challenging than you think, even for simple statements. For example, how should we pronounce an assignment statement like

$x = 5$? Is it “ x is 5”? Or “set x to 5”? Or “ x gets 5”? And what about an equality check? Is it “if x is is 5”? Or “if x is 5”? Or “is x is equal to 5”? As you can see, this could lead to tantalizing discussions.

3.12 Sensing and First Data

Matthew C. Jadud (Bates College – Lewiston, US)

License  Creative Commons BY 3.0 Unported license
© Matthew C. Jadud

My work broadly explores the behavior of of novice programmers and tools to support them in their learning. Recently, my students and I have been developing custom hardware for environmental sensing and extensions to Microsoft’s MakeCode online programming environment to support novices in quickly logging data captured from the world around them. Our goal is to enable the capture of small, personally relevant data sets that beginners can use when first learning to use code to work with data.

3.13 Giving Feedback and Hints in (Haskell/Java/...) Programming Tutors Based on Comparing Model Solutions to Student Solutions

Johan Jeuring (Utrecht University, NL)

License  Creative Commons BY 3.0 Unported license
© Johan Jeuring

Ask-Elle is a tutor for learning the higher-order, strongly-typed functional programming language Haskell. It supports the stepwise development of Haskell programs by verifying the correctness of incomplete programs, and by providing hints. Teachers can add programming exercises to Ask-Elle by providing a task description for the exercise, one or more model solutions, and properties that a solution should satisfy. A teacher can annotate properties and model solutions with feedback messages, and can specify the amount of flexibility allowed in student solutions. We calculate feedback using a variant of higher-order unification, extended such that it can deal with several more pragmatic aspects, such as the order of arguments of a function, or the order of declarations in a let expression.

3.14 Philosophical Concept Analysis in PL or SE or CSE or ...


Antti-Juhani Kaijanaho (University of Jyväskylä, FI)

License  Creative Commons BY 3.0 Unported license
© Antti-Juhani Kaijanaho

Concepts such as notional machines create a lot of confusion, because people use the same term in multiple different ways, and sometimes this difference in definitions is hard to spot. I argue that it is necessary to foreground the debate on concepts by having people explicitly state and defend their analyses (definitions), and for others who disagree to provide thoughtful counter-arguments. The goal might be a precise definition of the concept (in the classical style), or the replacement of a concept with a better one (in a Carnapian style). The result might be an agreed definition, but it also could be an agreement that the concept is incoherent, or actually is multiple concepts that need to be differentiated from each other.

3.15 Towards Algorithm Comprehension

Philipp Kather (Universität Münster, DE) and Jan Vahrenhold (Universität Münster, DE)

License  Creative Commons BY 3.0 Unported license
© Philipp Kather and Jan Vahrenhold

Comprehending and developing algorithms are very common activities in computer science and other studies. But what does it mean to comprehend an algorithm? Why are students creating flawed algorithms with correct proofs? We presented the current progress of a grounded theory study concerning algorithm comprehension to discuss this topic from a notional machines perspective.

3.16 Code and Cognition Lab


A. J. Ko (University of Washington – Seattle, US)

License  Creative Commons BY 3.0 Unported license
© A. J. Ko

My work contributes to the fields of computing education, human-computer interaction, and software engineering. My lab has recently focused on programming language learning, API learning, programming problem solving, machine learning literacy, and design literacy, as well as issues of diversity, equity, and inclusion in all of these topics. These discoveries building an evidence base for how to effectively and inclusively educate outstanding design- and data- literate programmers.

3.17 Language Levels

Shriram Krishnamurthi (Brown University – Providence, US)

License  Creative Commons BY 3.0 Unported license
© Shriram Krishnamurthi

Students don't program in one language; they program in several. Even through the course of a single book, ostensibly in a single language, the amount of the language they are exposed to keeps growing. This growth usually corresponds to an increase in complexity of the language's semantics. However, our IDEs rarely reflect this growth, presenting a monolithic language interface and leaving it to students to ensure they stay in the expected sublanguage. The DrRacket programming environment represents a rare exception, presenting a series of pedagogic languages, and including tools for building many more. The tools also vary with the language level – especially the Algebraic Stepper, which is a visualization of the notional machine. The corresponding book, *How to Design Programs*, also presents the notional machine also as a series of increasingly complex rules as the language grows; these rules are then manifest in the Stepper. In fact, different books choose to decompose the full language in different ways, and each can get the environment to reflect its chosen decomposition.

3.18 Explicit Programming Strategies


Thomas D. LaToza (George Mason University – Fairfax, US)

License  Creative Commons BY 3.0 Unported license
© Thomas D. LaToza

Software developers solve a diverse and wide range of problems, relying on programming strategies that they have learned. A programming strategy is a human-executable procedure for solving a programming task. We have developed a notation for writing strategies down explicitly in a program-like notation called Roboto. Using a strategy tracker tool, developers can follow a strategy step by step, as the computer keeps track of the next step and information they have collected executing the strategy. We've given explicit programming strategies to software developers and demonstrated that this representation enables developers to break their existing habits and work in new and more effective ways.

3.19 Conceptual Change & Knowledge in Pieces (KiP)

Colleen Lewis (Harvey Mudd College – Claremont, US) and Matthias Hauswirth (University of Lugano, CH)

License  Creative Commons BY 3.0 Unported license
© Colleen Lewis and Matthias Hauswirth

This workshop presented background information about theories within conceptual change, and particular details regarding the Knowledge in Pieces (KiP) perspective, created by Dr. Andrea A. diSessa (the PhD advisor of Dr. Lewis). The workshop presented three important aspects of KiP:

- (1) Typical uses and definitions of “mental model” ignore variations within a single student.
- (2) Learning involves learning to consistently use the “right” knowledge in different contexts.
- (3) Various knowledge fragments exist because they have been productive in some context.

For researchers new to the area, we also defined some frequently misunderstood terms: phenomenological-primitive (p-prim) and coordination class. The workshop concluded with a call to iteratively refine our understanding of CS learning by conducting research that takes into account students' moment-by-moment reasoning and is accountable to patterns of long-term conceptual change.

3.20 Concrete Notional Machines

Colleen Lewis (Harvey Mudd College – Claremont, US)

License  Creative Commons BY 3.0 Unported license
© Colleen Lewis

Abstraction is frequently mentioned as a core skill developed when learning programming, but computer science (CS) education rarely draws on education research focused on helping students build their understanding of abstraction. A particularly promising practice is known as concrete-to-representational-to-abstract, or CRA. Common practices for teaching addition are an example of CRA. CRA begins by introducing a physical (i.e., concrete) object. For example, this could be physical blocks that could be counted to add them together. Once

students are comfortable adding together sets of physical blocks, students could advance to solving the same problems given only a picture (i.e., representation) of the blocks. Once students are comfortable using only the pictures, students could advance to solving the same problems using only numbers (i.e., abstraction). If a student has trouble, they can always return to a previous representation. I have applied CRA to develop concrete memory models (i.e., the concrete), which then transition to drawn memory models (i.e., the representational), and ultimately Java code (i.e., the abstract).

3.21 Reference-point Errors: Slips? or Misconceptions of the Notional Machine?


Craig Miller (DePaul University – Chicago, US)

License  Creative Commons BY 3.0 Unported license
© Craig Miller

Novice programmers may mistakenly write code that references an object when the attribute of the object is intended, or vice versa. These errors are consistent with the use of metonymy, a type of figurative expression in human-to-human communication. Instead of misconceptions, the errors may be slips based on well-practiced habits of figurative communication

3.22 Notional Machines for Everyday Life


Andreas Mühling (Universität Kiel, DE)

License  Creative Commons BY 3.0 Unported license
© Andreas Mühling

Notional machines are typically seen as a way to allow learners to predict how a given program will execute. In the current discussion about “digital literacy” as a necessary qualification for all current and future citizens, the question arises how to explain how digital artefacts work without necessarily delving into programming. To this end, the idea of broadening the concept of notional machines to explain everyday phenomena (in particular also when considering communicating digitale devices) has been developed. The current state of the project was presented as a series of increasingly detailed abstractions of how digital devices work starting from a complete black box and ending somewhere above the von Neumann system of a machine. Each new level is introduced by a phenomenon that cannot be explained with the current level of detail. Future research will help in identifying the educational value of this concept and the optimal progression and level of granularity.

3.23 Making a Causal Diagram for Learning Programming

Greg Nelson (University of Washington – Seattle, US)

License  Creative Commons BY 3.0 Unported license
© Greg Nelson

Greg invited everyone to expand a simple causal diagram for learning programming, using sticky notes. The big idea behind making a causal diagram is that the feedback loops are the main determinants of system behavior (i.e. learning outcomes). Thanks to Ben du Boulay, Titus Barik, Rodrigo Duran, Kathryn Cunningham, Thomas LaToza, and Tom Ball for participating.

3.24 Pointer Concepts in C

Andrew Petersen (University of Toronto, CA)

License  Creative Commons BY 3.0 Unported license
© Andrew Petersen

Many introductory computing courses at the University of Toronto are built around frequent practice, supported by an online system that delivers online exercises and provides feedback on student submissions. In an early example of the type of analysis that can be performed from this data, we investigated student use (and mis-use) of pointers in their first week of exposure to pointer types in C. We defined a set of core pointer concepts to be covered in that week and then developed a pre- and post- assessment to identify which topics were most frequently mis-applied by students. We use the results of these questions to roughly order the concepts by difficulty. Additionally, we analyze student submissions to coding exercises, revealing inefficient behaviours students use to solve pointer problems and identifying the most common errors committed.

3.25 Semantics Tutorial


Joseph Gibbs Politz (UC – San Diego, US)

License  Creative Commons BY 3.0 Unported license
© Joseph Gibbs Politz

Semanticists use formalisms like grammars, relations, and trees to model the behavior of programs and programming languages. As they are a description of how programs evaluate, semantics are closely related to notional machines. This tutorial motivates semantics for programming languages with an example of syntactic scope in Python and substitution in Racket. It then goes on to show a worked example of a small-step, substitution-based semantics with evaluation contexts (in the style of Felleisen and Hieb) for a subset of Python.

3.26 Activity Theory

R. Benjamin Shapiro (University of Colorado Boulder, US)

License  Creative Commons BY 3.0 Unported license
© R. Benjamin Shapiro

Computing education research typically draws on theories from educational psychology and cognitive psychology. While useful, these theories, and their applications in computing education research, often fail to account for the situated, embodied, culturally-constructed, historically anchored, social, and materially-mediated nature of learning. I describe how activity theory can help us to attend to the practice of computing education in more nuanced and expansive ways. Here, practice refers to systems of teaching and learning, including how tools (like programming languages) are used within that practice, are designed with particular sets of values and practices in mind, and are also adopted based on sets of practices that may or may not be shared by researchers and designers operating in this area. I then draw on questions posed by Engeström to challenge the audience to consider

- (a) how the history of computing, computing education, and educational institutions shapes our present practice,
- (b) what tools and signs (e.g. programming languages or assessments) are available to different participants in the networks of computing education practice, and how they are used to construct the objects of our activity,
- (c) what contradictions exist within our activity systems, and
- (d) to thoughtfully consider what can and should be done to construct better systems of practice.

3.27 Stuff We Wish We Knew (About Notional Machines)


Juha Sorva (Aalto University, FI) and Otto Seppälä (Aalto University, FI)

License  Creative Commons BY 3.0 Unported license
© Juha Sorva and Otto Seppälä

We identified four areas with open questions related to notional machines. Dynamic visualizations of science concepts work best when designed for specific roles within a pedagogical approach and when students are taught representational competencies for reading the visualization, but there is insufficient research on these topics in notional-machine visualization. Little is known about how various increasingly common programming-language features – such as higher-order functions, type systems, type inference, and anonymous functions – should be attended to in notional machines. Studies of programming knowledge could be better informed by knowledge-in-pieces theories of conceptual change, which suggest a role for notional machines in integrating fragmented knowledge. The concept of notional machine requires further analysis and clarification, and it is unclear whether the term is helpful when disseminating research-based practices to teachers.

3.28 Revisiting Two Past Publications through the Lens of Notional Machines

J. Ángel Velázquez Iturbide (Universidad Rey Juan Carlos – Madrid, ES)

License  Creative Commons BY 3.0 Unported license
© J. Ángel Velázquez Iturbide

In this poster, I presented two past publications [1,2] in a slightly different way, through the lens of notional machines. In one work [1], three instantiations of recursion (namely recursion in grammars, in functional programs, and in procedural programs) were analyzed to understand varying difficulties of students in understanding them. The analysis identified their different representations of information and operational models, hypothesizing increasing complexity of their, let us say, notional machines. In a second work [2], we presented and evaluated a novel approach to enhancing students' understanding of recursion. We presented removal of linear recursion into equivalent, iterative code by means of a transformation scheme. In retrospect, we were explaining a part of a procedural notional machine (namely, recursion) in terms of another part of the same notional machine (iteration).

References

- 1 J.Á. Velázquez-Iturbide. “*Recursion in gradual steps (is recursion really that difficult?)*”. Proc. SIGCSE 2000, 310-314, DOI 10.1145/330908.331876
- 2 J.Á. Velázquez-Iturbide, M.E. Castellanos & R. Hijón-Neira. “*Recursion removal as an instructional method to enhance the understanding of recursion tracing*”. IEEE Trans. Education, 59(3):161-168, August 2016, DOI 10.1109/TE.2015.2468682

4 Working groups

4.1 Breakout Group 1


Geoffrey L. Herman (University of Illinois – Urbana Champaign, US) and Philipp Kather (Universität Münster, DE)

License  Creative Commons BY 3.0 Unported license
© Geoffrey L. Herman and Philipp Kather

This breakout group suggested a perspective on notional machines as a model between the source code and the actual machine. Models such as those in physics are only useful within certain bounds. They are half truths, excluding some aspects to be more useful in some contexts. Research questions related to the bounds of notional machines in various learning contexts, such as teaching multiple languages interleaved were developed. The need for instruments measuring the quality of students understanding of notional machines and the relevance of considering non-cognitive factors was also highlighted.

4.2 Breakout Group 2


Craig Miller (DePaul University – Chicago, US) and Franziska Carstens (Universität Münster, DE)

License  Creative Commons BY 3.0 Unported license
© Craig Miller and Franziska Carstens

This breakout session was shaped by the question “What do we need to know about notional machines that we don’t know yet?” The objective was to brainstorm on possible research questions, record them and discuss about appropriate study designs. During the discussion, the group pointed on the questions: what is the relation between notional machines and formal semantics, what is the language we use to describe a notional machine, and what is instructors practice on notional machines? To answer these questions, four different approaches were discussed. The first suggestion was to look at experts and collect data on explanations given by instructors. A second idea was to focus on the students execution of different possible notional machines and come up with an experimental control group design. For a third approach the group discussed about examining textbooks to identify presented notional machines and at least, they thought about taking a look at other areas (e. g. electrical engineering) and get an inside if and how notinal machines are used and presented there. During the whole session, the participants reflected on a domain sensitivity and asked if we would need different notional machines for different domains or if it is more a question of highlighting parts of one notional machine. In the course of the session, the participants increasingly used the term notional machine synonymous with lie. This perspective led to further considerations such as the impact of lies in a notional machine on students learning and the question of how much practice is required so that a simplified notional machine or lie becomes obsolete. In the end, the group agreed on taking a further look on instructors’ perspectives and formulated the research question “How much are instructors willing to lie to their students?” Under the assumption that every instructor has preferences and beliefs, the group thought about a survey-study to identify preferred ‘lies’ of instructors to their students and collect information on influences that may lead to possible preferences.

4.3 Breakout Group 3

Brett A. Becker (University College Dublin, IE), Neil C. C. Brown (King’s College London, GB), Paul Denny (University of Auckland, NZ), Rodrigo Duran (Aalto University, FI), Robert L. Goldstone (Indiana University – Bloomington, US), Antti-Juhani Kaijanaho (University of Jyväskylä, FI), Greg Nelson (University of Washington – Seattle, US), Carsten Schulte (Universität Paderborn, DE), Otto Seppälä (Aalto University, FI), and Steven A. Wolfman (University of British Columbia – Vancouver, CA)

License  Creative Commons BY 3.0 Unported license
© Brett A. Becker, Neil C. C. Brown, Paul Denny, Rodrigo Duran, Robert L. Goldstone, Antti-Juhani Kaijanaho, Greg Nelson, Carsten Schulte, Otto Seppälä, and Steven A. Wolfman

Breakout group number three reflected and generated research questions about previous approaches that in some manner used or were built on the concept of a notional machine and what instructional design was used to achieve the goal of such approach. The discussion topics generally fell into 3 categories:

- (a) How do we elicit learner mental models?

- (b) How do we change learner mental models?
- (c) How do we achieve near-term and long-term pedagogical goals?

These topics included: how mental models are used in practice; how to elicit students mental models and how they are connected to a notional machine presented by the instructor; how notional machines along with a concrete-to-formal continuum impact learning outcomes for different audiences; what are the perspectives of the CSEd community regarding notional machines; How to sequence notional machines and how to achieve transfer between programming languages, and which presentation form best suits novices. Greg's report back slides are here and are a quick synthesis of the clusters of RQs our group generated, and also include pictures of original ideation materials.

4.4 Breakout Group 4

Kathi Fisler (Brown University – Providence, US) and Kathryn Cunningham (University of Michigan – Ann Arbor, US)

License  Creative Commons BY 3.0 Unported license
© Kathi Fisler and Kathryn Cunningham

We considered the scope of notional machines. We agreed that a notional machine is a pedagogical tool, and it must explain the execution of programs. For systems that don't have programs (e.g. a cell phone in standard use), notional machines don't apply, and findings from HCI are likely more applicable. What is a minimal notional machine? IFTTT (If This Then That) is a platform where people can program different systems to interact with each other using very simple rules in "if-then" format. The notional machine to understand IFTTT seems quite limited, although the applications are powerful. This balance is possible since so much of the functionality is black-boxed. We believe a notional machine is a mediating artifact that attempts to reconcile the mental model of a student and a teacher. From this perspective, there are research questions about the way teachers interact with notional machines, the way students interact with notional machines, and the way the context of a topic or learning environment interacts with notional machines. We decided that one of the foundational research questions is how instructors use notional machines in practice. We proposed a collection of notional machine examples from a variety of instructors, to examine the different ways that instructors describe program execution to students.

4.5 Categorizing Notional Machines and their Representation or Visualization

Franziska Carstens (Universität Münster, DE)


License  Creative Commons BY 3.0 Unported license
© Franziska Carstens

The aim of this group was to develop a better understanding of the characteristics of notional machines. At the beginning of the session, the group had a short introduction about pattern language, given by Sally Fincher. During this talk, the participants got a small inside into The Engineer's Sketch-Book (<https://archive.org/details/engineerssketchb00barb/page/n17>) and discussed structuring principles and their importance. Afterwards, the group brainstormed

characteristics of notional machines and refined the result with concrete examples that originate directly from teaching practice. The group stayed with the plan to conduct an interview study after the seminar to collect further simplifying examples from teaching practice that are used to help students understand program execution or program state.

4.6 Instructional Design for Notional Machines

Barbara Ericson (University of Michigan – Ann Arbor, US), Robert L. Goldstone (Indiana University – Bloomington, US), Matthias Hauswirth (University of Lugano, CH), Antti-Juhani Kaijanaho (University of Jyväskylä, FI), Greg Nelson (University of Washington – Seattle, US), André L. Santos (University Institute of Lisbon, PT), and Anya Tafliovich (University of Toronto, CA)

License  Creative Commons BY 3.0 Unported license
 © Barbara Ericson, Robert L. Goldstone, Matthias Hauswirth, Antti-Juhani Kaijanaho, Greg Nelson, André L. Santos, and Anya Tafliovich

This breakout group generated a set of instructional design guidelines, as well as a library of examples of instructional designs.

4.7 Concept Analysis for Notional Machines

Antti-Juhani Kaijanaho (University of Jyväskylä, FI), Thomas Ball (Microsoft Research – Redmond, US), Markus Müller-Olm (Universität Münster, DE), and Juha Sorva (Aalto University, FI)

License  Creative Commons BY 3.0 Unported license
 © Antti-Juhani Kaijanaho, Thomas Ball, Markus Müller-Olm, and Juha Sorva

This ad-hoc breakout group discussed the need for a tradition of deliberate argumentation in support or against particular concept analyses, taking as a starting point Kaijanaho’s Onward 2017 essay “Concept analysis in programming language research: Done well it is all right”. We concluded that if someone believes another to be wrong, it is their obligation to respond with a counter-argument. It is, however, difficult to publish such arguments in academic forums, and we believe this needs to change. Right now, we can start by collecting current understanding of notional machines and by encouraging people to write position papers with deliberate argumentation.

4.8 Notional Machines for Everything

Joseph Gibbs Politz (UC – San Diego, US), Mark Guzdial (University of Michigan – Ann Arbor, US), and Philipp Kather (Universität Münster, DE)

License  Creative Commons BY 3.0 Unported license
 © Joseph Gibbs Politz, Mark Guzdial, and Philipp Kather

This breakout group discussed aspects of notional machines for parallel computing, reactive programming, reading mathematical proofs and javascript. Considering the audience the notional machine is taught to was the most important aspect when one develops a notional machine. A mental model of a notional machine, e.g. for serial computing, might be already developed, when a student engages in parallel computing. Events triggering the need for a

new notional machine were discussed as opportunities for introducing a new notional machine or extending the existing one. However, many languages used in industry allow behavior that is convenient for experienced programmers but too complex for a student to grasp at once. This is why the development of sub-languages and defining their notional machines in a way, that a student would be able to explain all behavior, would be an appropriate approach to teach those languages.

4.9 Notional Machines for Scratch and Python

Otto Seppälä (Aalto University, FI), Thomas Ball (Microsoft Research – Redmond, US), Titus Barik (Microsoft Research – Redmond, US), Brett A. Becker (University College Dublin, IE), Paul Denny (University of Auckland, NZ), Rodrigo Duran (Aalto University, FI), Juha Sorva (Aalto University, FI), and J. Ángel Velázquez Iturbide (Universidad Rey Juan Carlos – Madrid, ES)


License © Creative Commons BY 3.0 Unported license

© Otto Seppälä, Thomas Ball, Titus Barik, Brett A. Becker, Paul Denny, Rodrigo Duran, Juha Sorva, and J. Ángel Velázquez Iturbide

We originally set ourselves a goal to study and contrast the notional machines for Python and Scratch. A blog post by Greg Wilson “Is this a notional machine for Python?” (<http://third-bit.com/2018/04/12/notional-machine-for-python.html>) was used as a starting point for the discussion and a possible reference to create and contrast a Scratch version with. A notional machine can be suited for a specific audience and written to target a specific part of the programming language and the execution environment. Studying Wilson’s suggested notional machine for Python, we found it in many cases to be more generic in nature and describe features common to imperative languages in general – characteristics such as memory management and call stack behavior. This led to discussions about notional machines that could be used for a family of languages. Scratch, being a language that has been intentionally simplified for a younger audience, however was not found to have all the traits described in Wilson’s notional machine. Recreating, for example, a recursive version of a function calculating a factorial using Scratch was not possible as Scratch only allows user-made procedures without return values. (The function studied in the first meeting can be seen here: <https://tinyurl.com/dags-nm>) In our second meeting our goal was to start from the most minimalistic Python(-compatible) program imaginable, to consider the minimal language required to reason about the program and to come up with a sound and complete notional machine for this specific language. Our first program consisted only of a single assignment. The notional machine for this language had three rules explaining variables, values and assignment. We then augmented our program three times, each time using a new language feature and tried to find the minimal addition to the existing rules. This eventually led to a progression of notional machines each building on the previous iteration. One key outcome from the exercise was the additive design process itself.

4.10 Notional Machines and Simulation

Steven A. Wolfman (University of British Columbia – Vancouver, CA), Kathryn Cunningham (University of Michigan – Ann Arbor, US), Kathi Fisler (Brown University – Providence, US), Greg Nelson (University of Washington – Seattle, US), and Jan Vahrenhold (Universität Münster, DE)

License  Creative Commons BY 3.0 Unported license

© Steven A. Wolfman, Kathryn Cunningham, Kathi Fisler, Greg Nelson, and Jan Vahrenhold

This breakout group elaborated the use of notional machines as pedagogical tools during code writing, tracing and debugging and discussed how students develop a coherent mental model of those. There are clues that notional machine knowledge is not immediately helpful for code writing. If notional machine knowledge could somehow be integrated with higher-level, more abstract knowledge—such as programming plans and goals—notional machine knowledge and code writing knowledge may be brought together.

The breakout group suspects that the most fruitful goal for notional machines in students' code creation process is to foster a habit-of-mind of meta-cognition about the behaviour of the code they generate. Research questions going forward consider goals of instructors teaching notional machines, students' and experts' application of notional machines, pedagogical practices encouraging a productive habit-of-mind of simulating/verifying code and the use of documentation of abstraction created by students or instructors to elaborate their mental models of notional machines.

Participants

- Thomas Ball
Microsoft Research –
Redmond, US
- Titus Barik
Microsoft Research –
Redmond, US
- Brett A. Becker
University College Dublin, IE
- Neil C. C. Brown
King’s College London, GB
- Franziska Carstens
Universität Münster, DE
- Luke Church
University of Cambridge, GB
- Kathryn Cunningham
University of Michigan –
Ann Arbor, US
- Paul Denny
University of Auckland, NZ
- Brian Dorn
University of Nebraska –
Omaha, US
- Benedict du Boulay
University of Sussex –
Brighton, GB
- Rodrigo Duran
Aalto University, FI
- Barbara Ericson
University of Michigan –
Ann Arbor, US
- Sally Fincher
University of Kent –
Canterbury, GB
- Kathi Fisler
Brown University –
Providence, US
- Robert L. Goldstone
Indiana University –
Bloomington, US
- Mark Guzdial
University of Michigan –
Ann Arbor, US
- Reiner Hähle
TU Darmstadt, DE
- Matthias Hauswirth
University of Lugano, CH
- Arto Hellas
University of Helsinki, FI
- Geoffrey L. Herman
University of Illinois –
Urbana Champaign, US
- Felienne Hermans
Leiden University, NL
- Matthew C. Jadud
Bates College – Lewiston, US
- Johan Jeuring
Utrecht University, NL
- Antti-Juhani Kaijanaho
University of Jyväskylä, FI
- Philipp Kather
Universität Münster, DE
- A. J. Ko
University of Washington –
Seattle, US
- Shriram Krishnamurthi
Brown University –
Providence, US
- Thomas D. LaToza
George Mason University –
Fairfax, US
- Colleen Lewis
Harvey Mudd College –
Claremont, US
- Elena Machkasova
University of Minnesota –
Morris, US
- Craig Miller
DePaul University – Chicago, US
- Andreas Mühling
Universität Kiel, DE
- Markus Müller-Olm
Universität Münster, DE
- Greg Nelson
University of Washington –
Seattle, US
- Andrew Petersen
University of Toronto, CA
- Joseph Gibbs Politz
UC – San Diego, US
- André L. Santos
University Institute of
Lisbon, PT
- Carsten Schulte
Universität Paderborn, DE
- Otto Seppälä
Aalto University, FI
- R. Benjamin Shapiro
University of Colorado –
Boulder, US
- Juha Sorva
Aalto University, FI
- Anya Tafliovich
University of Toronto, CA
- Jan Vahrenhold
Universität Münster, DE
- J. Ángel Velázquez Iturbide
Universidad Rey Juan Carlos –
Madrid, ES
- Eugene Wallingford
University of Northern Iowa –
Cedar Falls, US
- David Weintrop
University of Maryland –
College Park, US
- Steven A. Wolfman
University of British Columbia –
Vancouver, CA

