Infrastructures of abstraction: how computer science education produces anti-political subjects

James W. Malazita & Korryn Resetar

DIGITAL CREATIVITY
https://doi.org/10.1080/14626268.2019.1682616

Infrastructures of abstraction: how computer science education produces anti-political subjects

James W. Malazita [a] and Korryn Resetar[b]

[a]Department of Science & Technology Studies, Rensselaer Polytechnic Institute, Troy, NY, USA; [b]Independent Scholar, San Francisco, CA, USA

ABSTRACT
Abstraction, defined in Computer Science (CS) as bracketing unnecessary information from diverse components within a system, serves as a central epistemological axis in CS disciplinary and pedagogical practices. Its impressions can be seen across curricula, syllabi, classroom structures, IT systems; and other dimensions of the epistemic infrastructure of CS (Malazita [Forthcoming]. "Epistemic Infrastructures, the Instrumental Turn, and the Digital Humanities." In *Debates in the Digital Humanities: Infrastructures, Institutions at the Interstices*, edited by Angel Nieves, Siobhan Senier, and Anne McGrail. University of Minnesota Press). As we will argue in this essay, abstraction in CS serves as an epistemic, cultural, and ideological wall to integrated critical-technical education, rather than as a bridge. Further, this wall is disguised as a bridge: the common language used across CS and the Humanities gives the impression that abstraction can be leveraged as a boundary object (Star [2010]. "This is Not a Boundary Object: Reflections on the Origin of a Concept." *Science, Technology, & Human Values* 35 (5): 601–617), as a point of connection among conflicting or incommensurable epistemic cultures (Knorr Cetina [1999]. *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge: MIT Press). Rather, computational knowledge practices leverage abstraction's homographic-ness, epistemically structuring collaborative efforts in anti-political ways. To illustrate the impacts of abstraction, this essay will introduce 'Critical CS1,' a hybrid pedagogical approach to teaching Computer Science through feminist and critical race theory. However, other components of the epistemic infrastructures of Computer Science, from curricular structure, to IT systems, to classroom culture, to the epistemic practices of coding itself, resisted these intervention efforts, and reproduced marginalizing effects upon students within the course.

KEYWORDS
Computer science education; critical pedagogy; feminist STS; abstraction; epistemic infrastructures

1. Introduction

Abstraction, defined in Computer Science (CS) as bracketing unnecessary information from diverse components within a system (Cook 2009), serves as a central epistemological axis in CS disciplinary and pedagogical practices. Its impressions can be seen across curricula, syllabi, classroom structures, IT systems, and other dimensions of the epistemic infrastructure of CS (Malazita, forthcoming). Though the term 'abstraction' is also commonly used across the humanities, social sciences, and engineering, its deployment in computer science pedagogy has particularly anti-political ideological

CONTACT James W. Malazita malazj@rpi.edu Department of Science & Technology Studies, Rensselaer Polytechnic Institute, 28 1st St, Apt 1, Troy, NY 12180, USA

consequences. As we will argue in this essay, though computer scientists characterize abstraction as a collaborative knowledge framework that affords interdisciplinary activity, abstraction instead serves as an epistemic, cultural, and ideological wall to integrated critical-technical education. Like the Coyote and the Roadrunner, this wall is disguised as a bridge: the common language used across CS and the Humanities gives the impression that abstraction can be leveraged as a boundary object (Star 2010), as a point of connection among conflicting or incommensurable epistemic cultures (Knorr Cetina 1999). Rather, computational knowledge practices leverage abstraction's homographic-ness, structuring collaborative efforts in anti-political ways.

To illustrate the impacts of abstraction, this essay will introduce 'Critical CS1,' a hybrid pedagogical approach to teaching Computer Science through feminist and critical race theory, developed and piloted at Rensselaer Polytechnic Institute (RPI). Critical CS1 is a modification of an existing course, 'Computer Science I' (CS1), which serves as the first required course for CS students at most universities. The approach interrogates traditional CS curricula, where discussions of structures of power and ideology are excised from technical classes, or taught as separate modules or 'computing and society' classes, if they are taught at all (Herkert 2005). In addition to integrated critiques of power and of algorithmic decision-making within core CS courses, Critical CS1 also aims to increase the recruitment and retention of marginalized students in CS. The course does this through highlighting how diverse ways of knowing are supported or resisted through epistemologies of computer science, and by introducing racial and gendered marginalization to students as both a political and epistemological problem.

While CS1 marks the formal introduction of the foundations of programming, often through the Python programming language, it is also foundational in that it establishes the epistemic, ideological, and political foundations of Computer Science for undergraduate students. These hybrid technical-epistemological foundations are heavily shaped by practices of abstraction, which when paired with vagueness of epistemic vocabulary used in CS and Artificial Intelligence knowledge cultures (Agre 1997), produces CS students as knowers who organize the world through excision. In other words, social phenomena are translated into technical systems through determining the 'relevance' of information to different components of an information architecture.

We argue that the epistemic infrastructures (Malazita, forthcoming; Murphy 2017) of CS education produce their students as ethical and epistemic subjects (Knorr Cetina 1999) whose technical and social identities are abstracted and split. To be a 'good programmer' becomes constructed as the ability to practice instrumental knowledge about abstracted, efficient programming techniques. Being a 'good person' means to be aware of 'broader' issues. And though many CS students become interested in practicing ethical coding, they also construct their technical competency as split from the social, political, and ideological world. Further, this production of split identity is doubly impactful for women, students of colour, queer and trans students, and other marginalized students, for whom body and identity is constantly reinforced as inseparable from their presence in any space, technical or not (Irigaray and Oberle 1985; Riley 2017). For these students, to 'become' a computer scientist means to reproduce the double-consciousness (DuBois 1965) of embodying knowledge practices and epistemological frameworks that counter their lived experience (Irani et al. 2010; Peters and Choi 2018; Carrigan 2018). The social/technical split encouraged by CS epistemic abstraction also produces a particular construction of what kinds of social and ethical interventions are needed and of where they are needed (Riley 2011). In short, principles of abstraction construct programming, and 'algorithmic' or 'computational' thinking more generally, as epistemically and politically neutral tools that only take political meaning once they are contextualized through the application. Similarly, ethical and social interventions in CS education become framed as valuable in application-centered classes, like data visualization or applied machine learning, but not in 'core' technical classes like Computer Science I. Without addressing CS's foundational epistemology of abstraction, critiques of the impact of computational systems on society are read by students as outside the purview of their domain expertise as programmers, undermining hybridized educational efforts before they begin.

2. Culture(s) of abstraction

We don't really think about the details of these programs; we just think of what they do for us. This is called an 'abstraction.' It allows us to think about a problem we are trying to solve without thinking about all the details of all the other systems we are depending on. Thinking in terms of abstractions is fundamental to computer science. (Stewart and Turner 2016, Computer Science I syllabus)

Abstraction in Computer Science is a defining feature of Object-Oriented Programming (OOP), a computational paradigm that operates through the proliferation and modification of 'objects.' Objects have predefined fields, or attributes, that can accept values, variables, and modifications, and which have predetermined internal and external relationships. Objects can be created and related either through 'Classes,' which operate as base object templates that new instances of objects inherit, or through 'Prototypes,' where new objects can inherit properties of prior objects declared as their prototype. A class-based approach, for example, might feature an 'Ice Cream' class with a 'Flavor' property, where instantiated objects of that class declare 'chocolate' or 'vanilla' as their 'Flavor.' A prototype-based approach, on the other hand, might feature 'Chocolate Ice Cream' and 'Vanilla Ice Cream' objects that each point to, and inherit the attributes of, an 'Ice Cream' object that serves as a prototype. Each approach promotes a tree-like ontology of objects that inherit attributes from their forebearers, and which are isolated from other object hierarchies.

There have been many critiques of the cultural values deemed inherent in OOP, in particular of its reproduction of formal ontologies and classifications of knowledge (Bowker and Star 1999), and for its 'bracketing' of objects from one another (McPherson 2014). Like the object-oriented status of a given programming language, however, these ideologies are difficult to pin down in any given software or programming environment. While object-oriented programming, somewhat ironically, is often thought of as a 'property' of coding languages (i.e. is a coding language object-oriented or not?), OOP is more coding practice than coding language. Most contemporary programming languages are multi-paradigmatic, in that they can be made object-oriented or not depending on programmer intent, style, and disciplinary convention. Similarly, multiple techniques that would all be considered 'object-oriented' can result in divergent computational architectures and ontologies. For instance, the same programming language, using similar OOP techniques, can be used to develop 'formal' or 'scruffy' ontological structures that feature divergent political and epistemological potentials, depending upon the goals and values of the programmer (Poirier 2017). The diversity of developers' epistemic regimes and their derived knowledge practices, then, are as politically impactful to software as the qualities of the language and operating system it runs on. OOP is a collection of particular kinds of knowledge practices within CS epistemic cultures and, given OOP's dominance as a programming paradigm, it is a collection of practices that permeates CS education, research, and broader knowledge work.

Though OOP can manifest in multiple ways, there are common knowledge practices that underpin them. The knowledge practice of 'abstraction' refers to the ability for programmes and programmers to operationally and epistemically excise content deemed as 'not relevant' to the function of the programme, class, or object. For example, abstraction determines that an object does not need to 'know' the intent, structure, or infrastructure of the digital system it is a part of. All it needs to know are the variables or values being passed to it, and what values its function is required to return to the broader system. Intentionally limiting knowledge of the broader system, programmers argue, makes both the code and the coding process more goal-oriented and more efficient.

Consider Thorben Janssen's illustration of abstractive coding practices in the 'real world':

> You need to know how to use your coffee machine to make coffee. You need to provide water and coffee beans, switch it on and select the kind of coffee you want to get.
>
> The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use. Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details. You just interact with a simple interface that doesn't require any knowledge about the internal implementation. (Janssen 2017)

There are similarities here with the epigraphic syllabus quote that heads the section. First is a privileging of information as 'need to know,' often constructed teleologically. In order for a user to efficiently use a coffee maker, their knowledge about and access to the system must be limited only to what is deemed necessary to complete their task. Likewise, their task must be defined in easily operationalized terms—the task of using a coffee maker is to make a cup of coffee; questions about the coffee's origin, the mining and manufacturing processes used to make the machine, the wages offered to the labourers who constructed and shipped the machine, the environmental impact the machine has, are beyond the scope of needed knowledge for the user. An efficient coffee maker does not burden the user with that information.

However, the coffee maker example is deceptive. Abstraction generally does not refer to sharing knowledge between the software and its user. Rather, abstraction involves hiding components of the system *from itself*. For example, if developers believe that one part of a system will stay relatively consistent (such as a process for managing memory cleanup), where another part of a system will regularly change (such as a process for passing user-input data from an ever-evolving user interface), the expected consistent component of the system will regularly be encapsulated. Encapsulation is an abstraction technique that utilizes information hiding principles to ensure software segregation; different components of the same system can be disallowed from 'seeing' one another. Though one component may pass values to and collect values from another, it will never have access to the processes that the component has applied to those values. It will also never have the capacity to modify the other component.

Programmers are taught to use encapsulation and information hiding in order to protect systems from themselves. Abstractive practices privilege stability and adopt a default stance of mistrust of new or changing components. From a teleological standpoint, this is a reasonable stance: if a developer knows that a component of a system is stable, i.e. that it is functioning as expected, then that stability should be shielded from new, potentially unstable components. It is, in theory, easier to find and fix bugs in a system when components are isolated from one another. By tracing 'when' bugs occur, i.e. when a system moves from stability to instability, developers are able to quickly figure which components are problem components and which are not. It is an onto-epistemological (Barad 2007) stance about how systems evolve and change, and about how knowledge is best operationalized and managed.

Abstraction is therefore not just the production of generalized rules or claims from specific instances; it is also the production of methods of the mapping of material and social worlds. That mapping has consequences: how we map social phenomena helps us determine where and what counts as points of intervention, disruption, or control (Clarke 2005). It is these knowledge practices that form the core of early-stage Computer Science education. This is not just an interpretive reading; while those outside of CS tend to construct CS education has heavily technical, instrumental, and, more broadly, on 'learning how to program,' CS faculty we have talked to tend to bristle at such characterizations. Commonly, when we asked Computer Science faculty what CS education is 'about,' we were told either 'algorithmic thinking' or 'computational thinking.' These were functionally interchangeable, and were generally categorized as a combination of:

- Breaking down complex problems into smaller, more tractable components
- 'Seeing through' the mess of reality in order to focus on 'only the details that are needed'
- Using step-by-step decision-making processes, generally by using logic gates or other formalizable decision trees, to solve a problem
- Finding an appropriate process will lead to appropriate solutions

The use of operationalized, teleological 'best epistemic practices' has been a common feature of programming practices for the past half-century, as documented by Tara McPherson in her analysis of UNIX development practices (2013). The first three listed here are inherent in the practice of abstraction: clearly defining hierarchical, interlocking, and inheritable relationships of components; stabilizing those relationships through encapsulation; and developing components to increase system stability. The fourth is self-fulfilling: the appropriate process is almost always an abstracted one. That algorithmic thinking principles dovetail with rhetoric used in scientific methods adds a level of rigour, and therefore cultural validity (Riley 2017), to abstractive epistemology.

When students in CS classrooms learn to code, they are also learning how to think about the social and material world. Those thought processes are then enacted and practiced through code and reproduced through curricula. These abstractive principles are used to construct the curricular structures, classroom cultures, and software ecosystems that produce the CS student experience, and thus the CS student. This production happens in two ways: by training CS students in modes of thought that bracket ethical or sociological content from technical concern, and by dis-enrolling and excising students who resist that mode of thought. The 'CS Student' is thus produced both as a subject position that learners occupy and also as a figure (Haraway 2018) that represents and enrolls certain political, ideological, epistemic, and identity positions.

To say that abstraction is foundational to of CS education is not to say that OOP is the only coding paradigm taught. Mid-level classes like Programming Languages are likely to have students experiment with OOP, other paradigms like functional, event-driven, and language-oriented, as well as hybridized approaches. Nor would we argue that OOP predates CS degree programmes. Both emerge in the United States in the late 1950s and early 1960s, with OOP branching out of developments of the Lisp programming family at MIT (McCarthy et al. 1960), and Purdue founding the first formal US CS department in 1962.

Rather, as Golumbia (2009) and McPherson (2012) articulate, the material and cultural dimensions of computing are co-produced, and cannot be separated or bracketed.

Abstraction and computational knowledge practices are particular ideological ways of knowing the world and our role in it, and, as Jasanoff argues, how 'know and represent the world … are inseparable from the ways in which we chose to live in it' (2004, 2). Similarly, although praxis is often spoken of as a way of introducing political content or practice into non-political, abstracted, or technical education (Walsh 2017), the co-production of the politics and practices of computing education highlights how political practice is always in these educational spaces, even if that practice is a practice of absence. If political content is included is itself a political decision. When teaching and learning, one cannot *not* do praxis.

These conditions are part of what Murphy and Malazita have separately labelled epistemic infrastructures (Murphy 2017; Malazita 2018a). Murphy highlights epistemic infrastructures as 'extensive arrangements of research and governance' (2017, 6) that do ontological and epistemological politics on a trans-national scale; in Murphy's case, the twentieth-century governmental economization and managerialization of populations. Malazita, in turn, highlights epistemic infrastructures as subjectively productive, in that their social and material systems that produce knowers, subjects, and epistemic objects (Knorr Cetina 1999), including how CS scholars construct the boundaries of their research practices, research objects, teaching, and disciplinary expertise (Malazita, forthcoming). Taken together, we can trace how the historical, ideological, and political implications of knowledge work are both causes and effects of the infrastructures they assemble.

Like all infrastructure, epistemic infrastructures are never static; they are always negotiated, built, and rebuilt (Star 1999), they change and are changed by their users (Velho 2017), and their boundaries and qualities are constructed in part by the positionality of the subjects and scholars who work through those infrastructures (Apperley and Parikka 2018). As we acknowledge that the design of scientific research and technological systems are never neutral, so too must we acknowledge that the design of curricula and educational cultures also always carry with them the assumptions, values, and knowledge frameworks of their designers (Knorr Cetina 1999). CS educational cultures and structures are major shapers of broader epistemic systems that underpin scientific and commercial development. The boundaries of disciplinary specialization—what counts as inside or outside legitimate disciplinary discourse—are always under negotiation. Computer Science, for example, includes a wide variety of discourse that on its face would seem to be 'outside' the boundaries of a technical discipline, but are in fact foundational to CS research and practice (ACM/AIS/IEEE-CS Joint Curriculum Task Force 2005, 31–36). Conversations about appropriate levels of abstraction when conceptualizing problems, or of the comparative beauty and elegance of algorithmic structures, are equally at home in computer science and humanities classrooms (Chandra 2014).

In the case of hybridized education, epistemic infrastructures produce the conditions of interdisciplinary teaching, as differing incentive structures and disciplinary training exponentially increase the labour of integrating content in non-modularized ways. Epistemic infrastructures do important identity work in higher education: they produce what faculty intuit are 'in bounds' and 'out of bounds' to address in their research and teaching, and also implicitly exclude students with conflicting epistemic identities and perspectives (Varelas, Martin, and Kane 2012). These exclusions contribute to a recursive, negative spiral: more epistemically diverse undergraduates are weeded out, leading to less epistemically diverse graduate students and faculty, thus contributing to an inertia of disciplinary epistemic rigidity (Rorty 1991) and narrowing diverse demographic participation.

From an epistemic perspective, the *ways* that content is scaffolded into an education system are as important as the content itself. As Donna Riley has argued, the construction of 'rigor' in STEM higher education often imagines social content as extraneous to 'core' technical content, and discussions of social structures in STEM classrooms are thought of as 'diluting' that core content (Riley 2017). Further still, this particular imagination of rigour marginalizes the perspectives of women, LGBTQ+ students, and students of colour (Riley 2017). The abstractive practices which co-produce the epistemic infrastructures of CS higher education reproduce the same epistemic violence we have seen across other dimensions of neoliberalism: a seemingly apolitical structure magnifies some identities and epistemic perspectives while marginalizing others (Allington, Brouillette, and Golumbia 2016). This marginalization occurs despite the lack of malicious intent or knowledge by faculty participating within, and indeed reproducing, structures of epistemic violence. The epistemic infrastructures of Computer Science, in fact, produce subjects as capable of knowing politics and justice in particular ways; ways that are often antithetical to critical perspectives on systems of power. These knowledge practices are not apolitical, in that they disregard the political and ethical dimensions of their work. Rather, they are anti-political, in that they acknowledge ethical and political dimensions, but in order to encapsulate and divest them from 'what counts' as within bounds of the computational aspects of sociotechnical systems and disciplines (Malazita, forthcoming).

3. Critical CS1 and unpacking the politics of CS epistemic infrastructures

Critical CS1 is designed as an interventive practice in the epistemic infrastructures of CS education, as a way of 'platforming' (Malazita 2018b) critiques of abstractive knowledge practices into those same practices. Our goal is to produce CS students who understand their knowledge practices as deeply technical and social, and who have the capacity to frame and critique the abstractive infrastructures of their own educational systems in order to challenge them. In short, we hope to use Critical CS1 to foster what Phil Agre called 'critical technical practitioners' (1997), or practitioners who simultaneously participate in and reconfigure the knowledge communities of which they are a part. In the following sections, we will detail the design and deployment of Critical CS1, as well as the cultural and infrastructural barriers that both students and faculty faced. These barriers, including classroom management and IT infrastructure required to participate in CS curricula at RPI, can be read through a critique of the cultures of abstraction that co-produce CS education.

The current version of Critical CS1, which has been under development since 2017, is a 'hack' of Computer Science 1, the first required course in RPI's CS major that teaches programming fundamentals through the Python language. Through lectures, lab assignments, homeworks with readings, and class discussions, we simultaneously teach programming concepts, Python techniques, and socio-political critique. The Critical CS1 syllabus outlines the following prompts for students:

(1) *Professional Identity and Problem Definition*: What gets defined as the boundaries of response-ability (Haraway 2008) for IT developers, in terms of broader social and moral imperatives, and in terms of our ability to materially address social conditions? Who belongs in the room at various stages during the design, development, and deployment process? To whom are developers accountable?
(2) *Structures of Power*: How do programmers participate in structures of power? How can well-meaning developers still reproduce systems of inequality and oppression? How do automated processes reproduce larger social biases, due in part to software authorship and the provenance of datasets?

(3) *Epistemology and Diversity*: What does it mean to understand Computer Science as a culture? What are the systems of knowledge and ideologies that guide Computer Science as a discipline? Where do these ideologies come from? How do they impact different people differently, and how might they alienate individuals with diverse identities and life experiences? How is your education training you to think, and what is it training you to think about?

Critical CS1 replaces the standard version of CS1 for enrolled students, with the long-term goal of Critical CS1 itself becoming the standard model of CS instruction. Pilot versions of the class were developed alongside two undergraduate students: both women, one a dual major in CS and Science & Technology Studies (STS) (Resetar), and one a former CS student who changed majors due to dissatisfaction with the social and epistemic cultures of CS (contributor Brookelyn Parslow). These students and an STS faculty member trained in computational design (Malazita) performed a close reading of prior CS1 syllabi to assess the programming and analytical thinking skills meant to be taught in each assignment, which were then refolded and socially contextualized into Critical CS1. As students in the Critical CS1 advance from using basic Python commands to building complex algorithmic structures, they are also introduced to more socially complex issues surrounding algorithmic decision-making and data.

Lectures for CS1 are 300–800 student large-format classes, with students further distributed into 15–40 person lab/recitation sections that meet once per week for two hours. Most of the content from Critical CS1 was delivered in via readings and discussion in one of these lab sections and through replacement assignments for the 8 major homeworks/projects given in Standard CS1. Each 'critical' homework assignment teaches the same technical content as a Standard CS1 assignment, but through examples, datasets, or discussions that also highlight political, epistemological, and historical dimensions of the assignment that otherwise would have been abstracted from view.

One assignment, for example, serves as an introduction to the technical and political aspects of sorting and algorithmic decision-making. While students are learning the technical foundations of dictionaries and search/sorting methods, they also read histories of immigration classification practices in New York during the early twentieth-century US-Irish Immigration, alongside a selection from Virginia Eubanks's *Automating Inequality* (2018), which tracks how algorithmic systems are used to classify, monitor, and punish the communities they are designed to serve. Using a dataset from the use of Bellevue Hospital as a sorting mechanism for 'undesirable' Irish immigrants in 1920s New York provided by Anelise Shrout, students practice using sorting algorithms to recreate the decision-making of immigration admittors during that era, including a case where pregnant Irish women are auto-classified as 'diseased.' The decisions, justifications, and technical practices used in 1920s New York are then compared to those used to police the poor and populations of colour in automated systems in Allegheny, PA, as documented by Eubanks. The goal of the assignment is to show students that politically valenced algorithmic decision-making is nothing new, and to have them see how historical oppressive practices are reproduced via new technologies upon new groups of people.

Each Critical CS1 assignment requires students to compose a 'README.txt' that documents their decision-making and development process as well as reflections from reading assignments and classroom discussions. As the semester advances, the technical and social dimensions of the README begin to blend more closely together. The final assignment in Critical CS1, for example, asks students to use the 'Fairness, Accountability, Transparency (FAT)' (Diakopoulos 2016) framework to create an 'unbiased' hiring algorithm for a large company. However, as students justified their programming decisions through the README, they began to see how all of their technical decisions, including what counted as technically 'fair' or 'accountable,' were subjective and shot through with structural power, and also reproduced systems of unequal access to education and social capital. The READMEs allowed students to discover, through their own practice, the powerful critiques of FAT and other 'unbiased' frameworks levied by STS scholars like Keyes, Hutson, and Durbin (2019) and Hoffman (2019). Students are given room to reflect upon their own findings and assigned readings during 'lab' sections each week. The course was also designed to subvert CS higher educational pedagogical structures by integrating insights from Friere's critical pedagogy (1968), including having students, faculty, and mentors work collaboratively, enabling students to challenge and change assignments, speaking frankly about the structure and design of the course, and inviting students to return in later iterations of the class as mentors and designers, thus allowing CS students to continually evolve their own educational apparatus.

The intersections of the subject position of the CS student, curricular structure, pedagogical culture, and abstracted epistemological framework worked to both integrate and resist the political valence of Critical CS1. This abstractive model of integrate-through-excision was most evident in how CS curriculum was modularized and encapsulated. Though we were aware of abstraction when first developing Critical CS1, we understood it more as an artefact of disciplinary specialization represented in curricular design, rather than as an epistemic lens through which pedagogical practice was viewed. The initial support from CS faculty for the course turned to well-intentioned resistance during the course construction practice. While Critical CS1 was designed to infuse political content throughout the entirety of CS1, we were quickly sidelined into a single lab/discussion section of 15 students, rather than integrated into the larger lectures. CS1 class examinations, which built off work done and examples used in Standard CS1, meant that students in our section needed to do twice the amount of homework: once to complete the Critical homework for their class, and once to complete the classical homework in order to prepare for tests. Course components, in other words, were abstracted and encapsulated to ensure system stability. The introduction of too much 'chaos' at once was seen as counter-intuitive to process-oriented algorithmic thinking and pedagogy. Of course, retaining stability in the pedagogical infrastructure also re-stabilizes the marginalizing impacts of that infrastructure.

Eleven students volunteered to enrol in the first Critical CS1 section, five of whom were women, and five of whom were students of colour. All spoke to us of their desire for more conversations and curricular intervention in CS classes about their identity experiences as marginalized persons in CS, and many told us that Critical CS1 provided them a space where their identities and ways of knowing were accepted and celebrated. However, they also were keenly aware that participating in this class meant that they had to do even more work than their peers, especially when prepping for exams that were not designed for them. Students enrolled in Standard CS1 also did not know how to 'make sense' of Critical CS1 students, with conflicting rumours flowing throughout the CS student body that Critical CS1 students were in the 'remedial' section, the 'advanced' section, the 'humanities-majors only' section, or the 'fake' section. Enrolling in Critical CS1 both alleviated some aspects of our students' marginalization while also introducing new ones.

This feeling of isolation of detachment was reinforced and reproduced through the IT systems that act as student management and auto-graders in class, as well. CS1 assignments are graded by Submitty, an in-house online homework submission server that checks students' code. Submitty grades functionally—it treats student code as textual output that can be checked against an 'ideal' approach to solving a programming problem, generally by requiring output to match an assumed correct answer or procedure. The platform, first created as a homework submission server, has become embedded within the pedagogical and curricular cultures of CS at Rensselaer. As its functionality has grown to accommodate enrolment management, grade distribution and reporting, a class forum, and course resources, Submitty has taken on a prominent role in almost every entry- and mid-level CS course. A non-trivial 'skill' taught in both Standard and Critical CS1 is learning to operate Submitty; students who don't will find it increasingly more difficult to complete even basic homework assignments in later courses.

Each homework submission is auto-graded for output only; i.e. did the student achieve the correct 'answer' for the coding problem, including exact white-spacing and punctuation. As Submitty allows for infinite re-submissions before a homework deadline, students tend to adopt a strategy of output-based programming, adjusting their code to meet the whims of white-space and style the software demands, rather than thinking through the epistemological patterns and social dimensions of the 'problem' they are given to solve. Students thus learn to code through the paradigm of Submitty: developing a programming solution that 'makes sense' or takes into account individual student knowledge or criticism of the problem is always subservient to the micro-demands of the auto-grader. While coding style is important, and debugging by anticipating the particular needs of other components of the system is vital for 'real-world' programming, Submitty's central place within our pedagogical framework changes the epistemic subjectivities of our students. Computational 'problem solving' becomes not about determining the multitude of influences and conflicts that a given programme may encounter and introduce to a socially vibrant world; it is about appeasing stabilized software.

Submitty as a facet of CS's epistemic infrastructure at Rensselaer actively undermines the critical and collaborative culture we attempted to foster through Critical CS1 assignments. For example, one homework assignment had students work together to develop short, branching stories that encourage them to connect their personal experiences with algorithmic systems and decision-making processes with class readings. While students were required to follow certain programming structures to reinforce the lecture material for the week (including learning the difference between 'for' and 'while' loops), the remainder of the assignment was relatively open ended. This became a problem when it came time to submit the homework on Submitty. Not only were these epistemic maps too large to be handed by the system, they also were incompatible with Submitty's autograder and plagiarism detector, which occupies a central algorithmic role in the processing of student assignments. In the auto-plagiarism detector, student code is compared against one another, and tested for 'solution' approaches. Submitty determines whether or not the ways in which a student decided to solve a problem is too similar to the strategies and styles used by another student, and will fail both students if it determines that their strategies are too similar. The interpersonal and managerial infrastructure surrounding plagiarism assumes guilt–if students contest the charge plagiarism, the burden of proof to determine that plagiarism didn't occur is on the students. CS Faculty, who have quite literally built Submitty, regularly refer to Submitty's ability to detect 'clever' plagiarism strategies, and adopt a default stance of believing the programme over students. It is fairly uncommon that Submitty accuses students of just copy-and-pasting code; rather, Submitty will charge that students 'worked together too closely' in figuring out a problem, and therefore did not demonstrate satisfactory mastery of the programming principle being taught.

Collaboration, discussion, knowledge sharing, and even helping other students is infrastructurally discouraged and punished. Rather, individual students are treated as instantiations of a figure of the CS student, who must master or inherent multiple attributes in order to pass into more advanced coursework. Again, we can see how this process does not come from a place of malice—there is a limited amount of time to teach a large amount of students, and designing a system that processes them as efficiently as possible is not an unreasonable solution. However, abstraction's incorporation-via-excision epistemological strategy, and the infrastructures that result from and reproduce it, reinforce the marginality of feminist, critical, and counter-hegemonic pedagogical strategies. And though Critical CS1 students generally reported feeling safer and having more agency in Critical CS1 than in Standard CS1, they also acknowledged that participating in the Critical version of the class made even more clear to them how much of their education was designed in a manner antagonistic to their life experiences.

4. Conclusion

The initial design of Critical CS1 focused on 'consciousness-raising' classroom interventions for hybridizing critical and technical pedagogy. In recognition that systemic marginalization is both rhetorically and structurally reproduced, it was also important to have these interventions occur within a curricular space that is commonly constructed as purely abstract and technical, rather than in a space for applied work or in a separate social sciences class.

However, other components of the epistemic infrastructures of Computer Science, from curricular structure, to IT systems, to classroom culture, to the epistemic practices of coding itself, resisted these intervention efforts, and reproduced marginalizing effects upon students within the course. This also left Critical CS1 students—especially those with marginalized identities—them in a state of epistemic tension. They were able to give voice to the systems that were marginalizing their ways of knowing and identities within Computer Science, while not providing the capacity to meaningfully change them. Moving forward, it is important for scholars of hybridized pedagogy to work with their students to help them identify the epistemological and structural conditions of their own educations, as well as to create multiple points of intervention and destabilization of the epistemic infrastructures that produce our students, and ourselves, as knowing subjects.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by National Endowment for the Humanities [AK-255350-17].

Notes on contributors

James Malazita is an Assistant Professor of Science & Technology Studies and of Games & Simulation Arts & Sciences at Rensselaer Polytechnic in Troy, NY.

Korryn Resetar is a designer, programmer, and independent scholar residing in San Francisco, CA. She earned her undergraduate degree from Rensselaer Polytechnic.

ORCID

James W. Malazita http://orcid.org/0000-0003-3206-3682

References

ACM/AIS/IEEE-CS Joint Curriculum Task Force. 2005. *Computing Curricula 2005, the Overview Report*. New York: ACM Press.

Agre, P. 1997. “Toward a Critical Technical Practice: Lessons Learned in Trying to Reform AI.” In *Social Science, Technical Systems, and Cooperative Work: Beyond the Great Divide*, edited by Geoffrey Bowker, Geoffrey Bowker, Susan Leigh Star, Les Gasser, and William Turner. Boca Raton, FL: CRC Press, Taylor & Francis.

Allington, D., S. Brouillette, and D. Golumbia. 2016. “Neoliberal Tools (and Archives): A Political History of Digital Humanities.” *LA Review of Books*, 1.

Apperley, T., and J. Parikka. 2018. “Platform Studies’ Epistemic Threshold.” *Games & Culture* 13 (4): 349–369.

Barad, K. 2007. *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*. Durham, NC: Duke University Press.

Bowker, Geoffrey C., and Susan Leigh Star. 1999. *Sorting Things Out: Classification and Its Consequences*. Cambridge: MIT.

Carrigan, C. 2018. “‘Different Isn’t Free’: Gender@ Work in a Digital World.” *Ethnography* 19 (3): 336–359.

Chandra, V. 2014. *Geek Sublime: The Beauty of Code, the Code of Beauty*. Minneapolis, MN: Graywolf Press.

Clarke, A. E. 2005. *Situational Analysis: Grounded Theory After the Postmodern Turn*. Thousand Oaks, CA: Sage.

Cook, William R. 2009. “On Understanding Data Abstraction, Revisited.” *ACM SIGPLAN Notices* 44 (10): 557–572. doi:10.1145/1639949.

Diakopoulos, N. 2016. “Accountability in Algorithmic Decision Making.” *Communications of the ACM* 59 (2): 56–62.

DuBois, W. E. B. 1965. *The Souls of Black Folk*. London: Longmans.

Eubanks, V. 2018. *Automating Inequality: How High-tech Tools Profile, Police, and Punish the Poor*. New York, NY: St. Martin’s Press.

Freire, P. 1968. *Pedagogy of the Oppressed*. New York: Continuum.

Golumbia, D. 2009. *The Cultural Logic of Computation*. Cambridge: Harvard University Press.

Haraway, D. J. 2008. *When Species Meet*. Minneapolis: University of Minnesota Press.

Haraway, D. J. 2018. *Modest_Witness@ Second_ Millennium. FemaleMan_Meets_OncoMouse: Feminism and Technoscience*. Oxfordshire: Ro utledge.

Herkert, Joseph R. 2005. “Ways of Thinking About and Teaching Ethical Problem Solving: Microethics and Macroethics in Engineering.” *Science and Engineering Ethics* 11 (3): 373–385. doi:10.1007/s11948-005-0006-3.

Hoffman, A. L. 2019. “Where Fairness Fails: Data, Algorithms, and the Limits of Antidiscrimination Discourse.” *Information, Communication & Society* 22: 900–915.

Irani, L., Janet Vertesi, Paul Dourish, Kavita Philip, and Rebecca E. Grinter. 2010. “Postcolonial Computing: A Lens on Design and Development.” CHI 2010, April 10–15, Atlanta, GA.

Irigaray, L., and E. Oberle. 1985. “Is the Subject of Science Sexed?” *Cultural Critique* (*1*): 73–88. doi:10.2307/1354281.

Janssen, T. 2017. “OOP Concept for Beginners: What is Abstraction?” *Stackify*. First Published November 23. Last Accessed July 29, 2019. https://stackify.com/oop-concept-abstraction/.

Jasanoff, S. 2004. *States of Knowledge: The Co-production of Science and the Social Order*. Oxfordshire: Routledge.

Keyes, O., J. Hutson, and M. Durbin. 2019. “A Mulching Proposal: Analysing and Improving an Algorithmic System for Turning the Elderly Into High-nutrient Slurry.” *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing systems*, April, alt06. ACM.

Knorr Cetina, K. 1999. *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge: MIT Press.

Malazita, J. 2018a. “Re: Configurations: A Shared Project for Literature and Science.” *Configurations* 26 (3): 269–275.

Malazita, J. 2018b. “Translating Critical Design: Agonism in Engineering Education.” *Design Issues* 34: 4.

Malazita, J. Forthcoming. “Epistemic Infrastructures, the Instrumental Turn, and the Digital Humanities.” In *Debates in the Digital Humanities: Infrastructures, Institutions at the Interstices*, edited by Angel Nieves, Siobhan Senier, and Anne McGrail. Minneapolis: University of Minnesota Press.

McCarthy, J., R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. 1960. *LISP I Programmers Manual*. Boston, MA: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory.

McPherson, Tara. 2012. “Why is the Digital Humanities So White? Or, Thinking the Histories of Race and Computation.” In *Debates in the Digital Humanities*, edited by Matthew K. Gold, 138–160. Minneapolis: Minnesota University Press.

McPherson, Tara. 2013. “U.S. Operating Systems at Mid-century: The Intertwining of Race and UNIX.” In *Race After the Internet*, edited by Lisa Nakamura and Peter Chow-White, 21–37. Oxfordshire: Taylor and Francis Group.

McPherson, T. 2014. “Designing for Difference.” *Differences* 25 (1): 177–188.

Murphy, M. 2017. *The Economization of Life*. Durham, NC: Duke University Press.

Peters, A. K., and J. O. Choi. 2018. “The Making of a Computer Scientist.” *XRDS: Crossroads, The ACM Magazine for Students – The Computer Scientist* 25 (1): 7–8.

Poirier, L. 2017. “Devious Design: Digital Infrastructure Challenges for Experimental Ethnography.” *Design Issues* 33 (2): 70–83.

Riley, D. 2011. “Engineering Thermodynamics and 21st Century Energy Problems: A Textbook Companion for Student Engagement.” *Synthesis Lectures on Engineering* 6 (3): 1–97.

Riley, D. 2017. “Rigor/Us: Building Boundaries and Disciplining Diversity with Standards of Merit.” *Engineering Studies* 9 (3): 249–265.

Rorty, R. 1991. *Objectivity, Relativism, and Truth: Philosophical Papers, Volume I*. Cambridge: Cambridge University Press.

Star, S. L. 1999. “The Ethnography of Infrastructure.” *American Behavioral Scientist* 43 (3): 377–391.

Star, S. L. 2010. “This is Not a Boundary Object: Reflections on the Origin of a Concept.” *Science, Technology, & Human Values* 35 (5): 601–617.

Stewart, C., and W. Turner. 2016. *Computer Science I Syllabus*. Last Accessed July 15, 2019. https://www.cs.rpi.edu/academics/courses/spring16/cs1/course_notes/lec01_intro.html?fbclid=IwAR35WRr3GrFHT2jFNGak9_7dcVKY0a52f7wYQPPBm1RnsDYzU_wBvm-moJY#types-of-problems-we-will-study.

Varelas, M., D. B. Martin, and J. M. Kane. 2012. “Content Learning and Identity Construction: A Framework to Strengthen African American Students’ Mathematics and Science Learning in Urban Elementary Schools.” *Human Development* 55 (5–6): 319–339.

Velho, R. 2017. “Fixing the Gap: An Investigation Into Wheelchair Users’ Shaping of London Public Transport.” Doctoral diss., University College London.

Walsh, B. 2017. “Collaborative Writing to Build Digital Humanities Praxis.” Digital Humanities Conference, August 2017, Montreal, CA.