Problematic and Persistent Post-Secondary Program Performance Preconceptions

Shriram Krishnamurthi
Brown University
Providence, RI, USA
shriram@brown.edu

Anika Bahl
Brown University
Providence, RI, USA

Benjamin H. Lee
Brown University
Providence, RI, USA

Steven A. Sloman
Brown University
Providence, RI, USA

ABSTRACT

Student conceptions about program "efficiency" shape their approach to programming and problem-solving. However, we know very little about the kinds of conceptions students have on entry into post-secondary education. In this paper we present the result of multiple iterations of a study where we ask students to rank programs on efficiency. We find students have several misconceptions across the iterations. We attempt to employ two standard techniques for puncturing people's illusions of understanding, but both have only limited success: students have strongly-held opinions despite their frequent errors. Post-secondary education about program efficiency needs to take much more account of students' pre-conceptions.

CCS CONCEPTS

• **Applied computing → Education**.

KEYWORDS

efficiency, misconceptions, illusion of explanatory depth, refutation texts

ACM Reference Format:
Shriram Krishnamurthi, Anika Bahl, Benjamin H. Lee, and Steven A. Sloman. 2022. Problematic and Persistent Post-Secondary Program Performance Preconceptions. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research (Koli 2022), November 17–20, 2022, Koli, Finland*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3564721.3564722

1 INTRODUCTION

Programmers often want efficient programs. To increase efficiency, they make both design and coding decisions. Therefore, it is important that their conceptions about efficiency be accurate. This is especially important because their conceptions factor into program design, as a recent paper [7, §3.2] shows. In particular, on grounds of efficiency, students rejected program structures that earlier work by Fisler [6] showed generally produced more correct programs.

These student opinions are reported as somewhat surprising, because most of the students had little prior computing experience, and the instructor stated that they do not discuss performance (and indeed dissuaded students from dwelling on it). In addition, as novices, the students have little understanding of implementation details, which can significantly alter performance (even changing asymptotic performance).

In this paper, we attempt to investigate this phenomenon more closely. We want to know what pre-conceptions students have about performance efficiency, and if these are wrong, whether standard techniques from other domains can help students overcome them. We design and apply instruments to students who are (mostly) starting post-secondary education but have had non-trivial secondary school computing experience. We do so because we believe the outcomes could inform what should and perhaps should *not* be covered in secondary school curricula, a topic that has not received much coverage from the perspective of the effects on post-secondary education.

Due to space limitations, we are forced to leave out many details. Instead, we have provided the material needed to judge the work and learn its important lessons. To aid reproducibility, every section has a corresponding appendix, which provides the full details. The paper provides the appendix as a supplemental document.

2 THEORETICAL FOUNDATIONS FOR INTERVENTIONS

The education literature makes clear [1, 10, 14] that direct instruction alone is unlikely to overcome misconceptions; activities that are more learner-centric are much more likely to be effective. Our work draws on two theories for tackling problematic beliefs.

The Illusion of Explanatory Depth. People generally do not understand how things work as well as they think they do. This was originally demonstrated by Rozenblit and Keil [13], who punctured people's illusion of understanding of how several common objects worked simply by asking them to explain. Subjects were first asked how well they understood how the objects worked and then to explain how they worked in as much detail as they could. For the most part, people were unable to construct anything resembling a reasonable explanation, revealing to themselves that they did not understand as well as they thought they had. Thus, when they were

Koli 2022, November 17–20, 2022, Koli, Finland

ACM ISBN 978-1-4503-9616-5/22/11…$15.00
https://doi.org/10.1145/3564721.3564722

again asked how well they understood how the objects worked, their judgments were lower, showing that subjects themselves realized they did not understand as well as they had thought they did. The authors referred to this finding as the illusion of explanatory depth (IoED). This phenomenon is seen across numerous domains.

Refutation Texts. Another promising avenue is the line of work starting from Posner et al. [12]. It presents a theory of conceptual change, at the heart of which is the *refutation text*. A refutation text tackles the misconception directly, providing a refutation for the incorrect idea. A recent study [20] showed their effectiveness in physics; we use this study's guidance in the design of our refutation text.

3 RELATED WORK

We know of only two papers that are closely related to our work. In a "commonsense computing" paper, McCartney, et al. [11] discuss algorithm efficiency. Students are given a problem and two semantically equivalent solutions, which they must choose between. Their responses are coded on multiple dimensions, the most relevant to us being how they *interpreted* the "fewest". These are clustered into conventional algorithm analysis categories such as "better worst-case" and "better average-case". Thus, there are many differences between this work and ours (it is over an abstract solution, not code; it focuses on conventional computational complexity; and it has no intervention). Still, it provides useful insight into how students think about computational complexity before formal training.

Most closely related is work by Gal-Ezer and Zur [8], who study late secondary-school students learning computer science. Their work predefines "efficiency" to be in terms of number of executed instructions. Based on work by Stavy and Tirosh [15], it hypothesizes that students will be thrown off by shallow syntactic criteria (e.g., program length). Its main study instrument is pairs of programs that are semantically identical but that vary syntactically according to these criteria. These are set up to trigger the hypothesized misconceptions, and the paper confirms that students do indeed exhibit these mistakes. (Similar findings are echoed in our work.)

Our work differs from this work in a few important ways. First, we do not fix what efficiency means; rather, we wish to learn how students have conceptualized it. Second, we use feedback from one iteration to enrich our study on the next round. Third, we choose a linguistic setting where students are much less likely to have experience and actual knowledge. Fourth, we choose programs that are actually equal or cannot be told apart with the given information. Finally, and most significantly, we create interventions around a collection of techniques well-grounded in cognitive science and educational psychology to *overcome* students' misconceptions, and evaluate the utility of those methods.

Additional but less-related work is discussed in appendix A.

4 STUDY CONTEXT

Our study was conducted at a highly-selective private university in the USA. The student subjects took an accelerated introduction to computer science, which covers much of the material of the first-year sequences at the university. Though self-contained and open to everyone, the class was primarily designed to provide an intensive experience for students with prior computing experience. Students therefore placed into it through a month-long placement process in the summer, during which they were expected to read *How to Design Programs* [4] and implement a variety of exercises (while obtaining assistance through an on-line message board). Once the semester began, students learned big-O time analysis and applied it to a variety of algorithms over lists, sets, trees, and DAGs, working up to basic graph algorithms. Students primarily programmed functionally.

5 STUDY ORGANIZATION

Our study was conducted over two separate years. The Summer 2020 data are labeled S20. We already saw the problematic phenomena that are at the heart of this paper. Our intent was to give students a refutation text and perform the study again. However, due to the stresses of virtual instruction and COVID, we were already trying to reduce work in the course, and hence opted to not perform a second round. In Summer 2021 (labeled S21-1), we conducted the study the second time. This study largely confirmed our earlier findings. On this iteration, we presented our students with the refutation texts during the course (in the Fall semester), and asked them to do the study one more time (but, because it was near the end of the semester, the second administration was made optional). These data are labeled F21-2.

6 STUDY POPULATION

In 2020, a total of 172 students completed the study. In 2021, 60 students contributed to S21-1 and 31 students to F21-2. The sharp discrepancy in numbers across the years is because of the way the institution restructured learning due to COVID-19. The lower number in F21-2 is because participation was left optional due to being late in the semester. We did not collect demographics for F21-2 to reduce student burden and because this group is a subset of S21-1.

Most students were new to post-secondary education. In 2020, due to a COVID-induced restructuring of courses, 78% were incoming, 14% had finished up to one year, and 8% had finished more than one year. In 2021, more typically of the course, 95% were incoming.

Most students in the study had prior computing background. The percentages below are slightly approximate because in some cases students wrote in answers that required interpretation. In terms of US Advanced Placement computer science, 37% in 2020 and 33% in 2021 had *not* taken an AP exam, but it is worth noting that several students may not have had the opportunity (due to being international, studying at a school that did not offer it, etc.). Over half of all respondents in both years had taken the AP CS A exam, which is considered more programming-oriented than AP CS Principles. Across the years, over 40% indicated knowledge of Web/JavaScript, over 45% a block-based language, over 68% Python, over 71% Java, over 26% C/C++, over 23% a statistical language, and 8% a Lispy language. Very few (3-4%) reported having no prior programming background at all.

In short, the population is mostly students who have had a non-trivial amount of prior computer science, and are hence being exposed to attitudes before entry. Students were asked where they had "gotten opinions on program efficiency from". The leading answers

(not mutually exclusive) were past teachers (52% in 2021, 33% in 2022), Web discussion sites (42% and 30%), classmates and friends (24% and 13%), and family (7% and 5%). In contrast, 22% and 43% respectively said "I don't really have any".

7 STUDY PROGRAMMING LANGUAGE

All our studies were conducted using programs written in the Racket programming language [5]. The placement process required students to use Racket alongside the course book [4]. At the point of S20 and S21-1, students had used Racket for about four weeks. After this, they did not see Racket again, so F21-2 students were referring back to a language they had not seen in over three months.

The use of Racket is not entirely incidental. It is already used in part because of its connection to the aforementioned book, but also because most students don't know it coming in: very few students (section 6) had any prior exposure to it (note that the 8% includes several Lispy languages). Therefore, this avoids giving most students a significant advantage over the others. Similarly, it offers numerous advantages in the context of this study:

- Because of their lack of prior exposure, they are unlikely to have strong prior knowledge about it. A vastly more popular language, like Java or Python, would have many more confounding factors based on what students had been taught previously.
- Students have little to no prior exposure to functional programming, and both the placement and course were primarily functional. Given that functional programming is generally perceived to be "different" from imperative or object-oriented programming, this would likely reduce the pre-conceptions they applied to Racket.
- Racket is syntactically distinctive (with a parenthetical, Lisp-inspired syntax), so it is unlikely to remind them too closely of languages they had programmed in before, which could otherwise have had some confounding recall effects.

Arguments for not using a made-up pseudocode are provided in appendix B.

8 STUDY INSTRUMENTS

Students were given three sets of *behaviorally identical* programs, followed by questions designed to learn about student perceptions. The programs are shown in fig. 1, fig. 2, and fig. 3. The programs satisfy different purposes. The first set examines student beliefs about fine-grained behavior. The second checks their impression about built-in functions. The third does the same, but specifically for built-in higher-order functions.

What matters for the purpose of this study is that these programs are effectively also all identical (or at least not reliably distinguishable) in terms of their execution time. For the benefit of the reader not familiar with Racket or performance considerations, a full explanation of these program texts is provided in appendix C.

In general, students were given sets of programs and asked to pair-wise indicate which they thought was "more efficient". In S20 they saw only sets 1 and 2, and set 1 had only (A) and (B). S21-1 and F21-2 used all the programs.

After making their initial estimate, students were asked to explain their choices. In S20, they were told to "take into account the transformations a compiler performs on code and the optimizations an architecture performs during execution that would apply in this setting", with the hope this would trigger their IoED. After they had done so, they were asked to rate their programs again, with the hope we would see results from this triggering. When we saw that the S20 students ignored the details of this prompt, in S21-1, we instead gave them a concrete list of 18 different compiler optimizations, most of which they (as expected) did not know, to more directly trigger their IoED.

For F21-2, instead of using IoED, we used refutation texts. These were shared with them right after the conclusion of S21-1, and again before F21-2, and also linked to the study instrument. Below we give one illustrative example of a refutation text entry:

> **Belief [LENGTH]:** A line of code takes a single unit (constant) amount of time to run.
> **Status:** Not necessarily true!
> **Explanation:** It depends on what code is on that line. Think about the most extreme case: you can write an infinite loop in one line; do you think an infinite loop finishes in one unit of time? Actually, it doesn't finish at all! In short, fewer lines of code don't necessarily run faster than more lines. A function call on that line means it could take a long time.

Full details of how the study versions differed are in appendix C, and the full refutation text is in appendix J.

9 ANALYSIS METHODS

The program comparisons are easy to summarize directly, and to compare against ground truth. However, two questions—one on their conception of efficiency, and the other on their explanation for their program comparisons—asked for free-form text. We therefore generated rubrics for assessing these. In both cases, two authors revised the rubric and computed Cohen's κ [2] for inter-rater reliability. The rubric for efficiency comparisons is shown in fig. 4 and obtained a κ of 0.88 after two iterations. The rubric for program textual explanations is shown in fig. 5 and obtained κ of 0.839 after five iterations. More detail about the creation of the rubrics is given in appendix D.

10 FINDINGS: CONCEPTIONS ABOUT "EFFICIENCY"

Figure 4 shows both the rubric and counts of student conceptions about efficiency. Students largely associate efficiency with running time, and to a lesser extent with the number of steps and with space usage. Students who referenced correctness did so in the context of improving performance (i.e., making a program faster without changing its correctness). Some students also referred to other syntactic criteria such as readability and writability. A few referenced power consumption. Appendix E discusses some of the differences (e.g., why two months of the course could have impacted, for instance, the drop in STEP and growth in TIME), but in general these numbers are quite instructive.

```
(A)
(define (f x)
 (cond
   [(empty? x) P]
   [(empty? (rest x)) Q]
   [else R]))
```

```
(B)
(define (f x)
 (cond
   [(empty? x) P]
   [(cons? x)
    (if (empty? (rest x))
        Q
        R)]))
```

```
(C)
(define (f x)
  (cond
    [(empty? x) P]
    [else
      (if (empty? (rest x))
          Q
          R)]))
```

Figure 1: Program Set 1

```
(A)
(define (len l)
  (cond
    [(empty? l) 0]
    [(cons? l) (+ 1 (len (rest l)))]))

(define (f g)
  (loop ... (len g) ...))
```

```
(B)

(define (f g)
  ;; length is built-in and
  ;; produces the same answer as `len`
  (loop ... (length g) ...))
```

Figure 2: Program Set 2

Suppose `L` is bound to a list of numbers:

```
(A)
(define (sum l)
  (cond
    [(empty? l) 0]
    [(cons? l) (+ (first l) (sum (rest l)))]))

(sum L)
```

```
(B)
(foldr + 0 L)
```

Figure 3: Program Set 3

Code	Explanation	S20	S21-1	F21-2
STEP	Mentions computational steps	40.2%	25.8%	11.3%
TIME	Mentions time/speed	66.4%	78.8%	87.1%
MEM	Mentions memory	26.5%	40.9%	58.1%
CORR	Mentions correctness	10.7%	12.1%	6.5%
CODE	Mentions amount of code	13.7%	4.5%	3.2%
MISC	Mentions some other interesting point	17.0%	18.2%	25.8%
READ	Mentions readability or comprehensibility	8.0%	0.0%	12.9%
WRIT	Mentions how easy the code is to write or otherwise work with	5.1%	1.5%	8.1%
POW	Mentions processing power, computational power, or energy	3.6%	6.1%	17.7%

Figure 4: Rubric for Student "Efficiency" Conceptions

11 FINDINGS: PROGRAM RANKINGS

We have numerous tables (nine in all) that give the detailed findings; they are all given in appendix F. However, these are not only not necessary, they may also obscure the central message, which we summarize here.

Keep in mind that the programs are essentially identical (section 8), but also that students do not know much about them (section 7). As a result, there are basically two answers that we expect from correct student rankings: that the programs are Equal or that the students don't know (IDK, short for "I don't know").

In contrast, student responses were uniformly off and highly problematic. In S20, the *highest* Equal response was 29% and *highest* IDK was 20%. Worse, the written activity, which should have triggered their IoED, had no impact.

In S21-1, their initial answers (with one exception, noted below) were similar: the highest Equal was 31% (but as low as 11%) and highest IDK was 20% (but as low as 0%). The only exception

Code	Explanation	S20	S21-1	F21-2
BIGO	References big-O when evaluating program efficiency	4.4%	17.7%	1.8%
BADO	Makes mistaken and/or fallacious big-O arguments	0.8%	5.4%	0.0%
OKAY	Big-O rationalization is correct (though it may be weak) if we were to take their assumptions to be true	3.6%	12.3%	1.8%
STEP	Considers number of operations (implicitly or explicitly) when evaluating program efficiency	42.8%	48.5%	25.4%
BLTN	Evaluation of efficiency relies on assumptions about builtin implementations	34.3%	67.7%	50.0%
WYSI	Assumes that source code corresponds directly to executed code without potential compiler modification(s)	41.5%	73.1%	50.9%
COMP	Evaluation of efficiency relies on assumptions about compiler behavior	28.8%	6.2%	3.5%
RCKT	Specifically mentions Racket in evaluation	4.4%	2.5%	16.7%
ELGN	Used style, succinctness, elegance, or "follow-ability" of a solution to assess its efficiency	11.6%	12.5%	4.4%
UNKN	References lack of knowledge with regards to builtin implementations and/or compiler behavior	13.7%	24.6%	51.8%

Figure 5: Rubric for Student Efficiency Explanations

was program pair 1(A)–1(C) (which was only introduced in 2021), which is exactly the same program after desugaring. 63% saw it as identical, but 35% did not; in particular, 26% were misled by the longer syntactic length (echoing [8] and [15]). Students were then asked to indicate which compiler optimizations they knew. 80% knew none; only two were chosen by three students. That is, most students admitted to little or no knowledge of the inner workings of programming language implementations. Despite this, when asked to rank the same programs again, their results barely shifted; in fact, the highest IDK went *down* to 14%, and even the Equal for pair 1(A)–1(C) went *down* to 57%. This is the opposite of what we would expect if IoED were triggered. Students were also asked to list their confidence on a 7-point Likert-like scale. Again, after being confronted with a long list of compiler optimizations that they indicated they did *not* know, students seem to have *increased* in their confidence in assessing efficiency.

In F21-2, we see much greater rates of Equal (23–58% for the rest, and 71% for 1(A)–1(C)) as well as of IDK (16–32%). There are many factors that may explain this: a semester of sophisticated computer science; extensive use of big-O (which, though they were told to not use in this study, may have lurked in their assessment, resulting in greater Equal scores); and of course the refutation texts. We have not attempted to tease apart these factors. Still, it is worth noting the bottom-line: *students confidently perceive all sorts of performance differences that don't exist*!

12 FINDINGS: RANKING EXPLANATIONS

Recall that students were also asked to provide written explanations for their rankings. Figure 5 shows both the rubric and counts of responses. Counts are aggregated across all pairwise comparisons for simplicity. While this loses some precision, our goal here is to study general phenomena.

We find the following points most significant. Unsurprisingly, non-trivial numbers of students "count steps" for evaluating program efficiency, but that reduces dramatically in F21-2. Our best explanation for this is that they have spent a whole semester in functional programming, where programs have non-trivial expressions—and often just one in a function body—rather than "lines" of imperative code. Nevertheless, many of them have a "what you see is what you get" (WYSI) interpretation; few of them seem to understand that the compiler is capable of having a small or even large impact on the generated program, even when confronted with a long list of alien terms. They are nevertheless quite confident about small syntactic differences (like between 1A and 1B) having a meaningful effect. Relatively few students use non-performance measures like elegance when analyzing efficiency, but more do than one might initially expect.

Again, there is much more one can examine and discuss. In particular, there are salient figures in the table above that could benefit from further explanation. Appendix G provides these.

13 THREATS TO VALIDITY

This work naturally has many threats to validity. We summarize them below, with elaboration in appendix H.

In terms of *internal validity*, we have notable differences in populations across years, and only a subset of S21-1 also took F21-2. These make some comparisons across these instances tricky. In particular, having similar populations might alter some of the ratios: seemingly problematic differences might disappear, while seeming non-differences may now stand out. We also showed students the *same* pairs of programs, which could cause view entrenchment; isomorphic programs may help, though they may also cause other issues (as seen, for instance, in variants [3, 16] of the Wason selection task [19]), making the results hard to reason about. We have also sampled students on very few programs.

There are many factors that affect the *external validity* of our findings: choice of language, non-standard pedagogy (section 4), student population, etc. The fact that many of our students have completed (and probably done well) on AP-like exams might make them overconfident.

In terms of *ecological validity*, the "programs" in our instruments are not actually *programs* but rather templates with placeholders. When given real and full programs, students would naturally simply be able to run and measure them, especially if their primary efficiency concern is a physical quantity such as time or memory.

14 DISCUSSION

We conclude with some discussion points. We provide additional discussion points in appendix I.

Transfer. We wonder about the impact of students' experiences with the world they live in and the material they learn in courses. Consider, for instance, a typical algorithmic topic: binary search, and a proof that logarithmic complexity is a lower bound. Yet many students have spent a decade or more using search engines, which clearly respond in constant time even as the Web keeps growing. How do students reconcile these phenomena? Do they fail to even make a connection (a form of failure to transfer [17])?

Is Counting Steps Harmful? Students are pressured to count steps at a potentially unhelpful level. For instance, as of this writing, the very earliest Code.org curriculum—"Pre-reader Express" for Ages 4–8—will, as early as Puzzle 7 in Lesson 2, give feedback about extraneous (but functionally harmless) blocks: e.g., *Congratulations! You completed Puzzle 7. (However, you could have used only 3 blocks.)* This is the only criterion (beyond correctness) applied to programs. We believe it is unhealthy to prematurely instill poor optimization ideas, and know of no research supporting it. Similarly, the Ask-Elle tutor [9] pushes students away from a quadratic solution even when a student is still learning the basics.

Absolving the Sin of Poor Efficiency. Anecdotally, along the lines of what [7] report, we have seen students exhibit strong resistance to certain program structures that they perceive as "wasteful" (in particular, ones that traverse a compound datum—such as a list—more than once). Curiously, we have seen that giving students some information about compiler optimizations such as deforestation [18] (which merges multiple traversals into one) significantly changes their perception. On its own, this is not surprising. However, this change persists even when they are told that the compiler they are using does *not* perform this optimization! This suggests that their performance conceptions are actually even more complicated than this paper's analysis suggests.

The Confidence to Not Know. We find it notable that students seem to generally be reluctant to say that they do not know an answer. There are many phenomena that could be at work here. Students in general may not be accustomed to their educational system giving them questions that don't have clear answers, and thus may have felt some pressure to guess. (However, this should have been reflected in lower confidence numbers, which it did not! That suggests this is only a partial explanation.) It is also possible that our student population is stubborn in all respects, not only on program efficiency, and hence somewhat impervious to IoED. Their strong high-school preparation may have also given them inflated confidence. Finally, they may also subtly mirror the university's admissions processes, which may (inadvertently) select for students who express confidence, even when misplaced!

ACKNOWLEDGMENTS

We are grateful to Usama Naseer, Theophilus Benson, and Malte Schwarzkopf for useful discussions about performance measurement in systems. This work was partially funded by US National Science Foundation grant DGE-2208731.

REFERENCES

[1] M. Cakir. 2008. Constructivist Approaches to Learning in Science and Their Implications for Science Pedagogy: A Literature Review. *Intl. J. Env. & Sci. Ed.* 3, 4 (2008), 193–206.
[2] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20 (1960), 37–46.
[3] Leda Cosmides and John Tooby. 1992. Cognitive Adaptions for Social Exchange. In *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*, Leda Cosmides, John Tooby, and Jerome H. Barkow (Eds.). Oxford University Press.
[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs.* MIT Press. http://www.htdp.org/
[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. In *Communications of the ACM.*
[6] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *SIGCSE International Computing Education Research Conference.* 35–42. https://doi.org/10.1145/2632320.2632346
[7] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computer Science Education.*
[8] Judith Gal-Ezer and Ela Zur. 2004. The efficiency of algorithms—misconceptions. *Computers & Education* 42, 3 (2004), 215–226. https://doi.org/10.1016/j.compedu.2003.07.004
[9] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27 (2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x
[10] J. Longfield. 2009. Discrepant Teaching Events: Using an Inquiry Stance to Address Students' Misconceptions. *Intl. J. Teach. and Learn. in Higher Ed.* 21, 2 (2009), 266–271.
[11] Robert McCartney, Dennis J. Bouvier, Tzu-Yi Chen, Gary Lewandowski, Kate Sanders, Beth Simon, and Tammy VanDeGrift. 2009. Commonsense Computing (Episode 5): Algorithm Efficiency and Balloon Testing. In *International Workshop on Computing Education Research.* 51–62. https://doi.org/10.1145/1584322.1584330
[12] G. J. Posner, K. A. Strike, P. W. Hewson, and W. A. Gertzog. 1982. Accommodation of a Scientific Conception: Toward a Theory of Conceptual Change. *Sci. Edu.* 66, 2 (1982), 211–227.
[13] Leonid Rozenblit and Frank Keil. 2002. The misunderstood limits of folk science: an illusion of explanatory depth. *Cognitive Science* 26 (2002), 521–562.
[14] L. Savion. 2009. Clinging to discredited beliefs: The larger cognitive story. *J. Schol. of Teach. and Learn.* 9, 1 (2009), 81–92.
[15] Ruth Stavy and Dina Tirosh. 1996. Intuitive rules in science and mathematics: the case of 'more of A – more of B'. *International Journal of Science Education* 18 (1996), 653–667. https://doi.org/10.1080/0950069960180602
[16] Keith Stenning and Michiel van Lambalgen. 2008. *Human Reasoning and Cognitive Science.* MIT Press.
[17] Edward L. Thorndike and Robert S. Woolworth. 1901. The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review* 8 (1901).
[18] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73 (1990), 231–248.
[19] Peter Cathcart Wason. 1966. Reasoning. In *New Horizons in Psychology I*, B. M. Foss (Ed.). Penguin.
[20] Kristin M. Weingartner and Amy M. Masnick. 2019. Refutation texts: Implying the refutation of a scientific misconception can facilitate knowledge revision. *Contemp. Edu. Psych.* 58 (2019), 138–148.