Student Understanding of Aliasing and Procedure Calls

Preston Tunnell Wilson
Brown University
ptwilson@brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Kathi Fisler
Brown University/WPI
kfisler@brown.edu

Abstract

Procedure (or method) calls are a basic computation mechanism found in virtually every language. A procedure call may or may not create *aliases* for parameters. Understanding aliasing is critical for comprehending how programs will behave, with impact on other concepts such as parallelism.

In this paper we study the awareness and descriptions of aliasing behavior in two college-level audiences. The paper measures their understanding of aliasing, analyzes their written explanations of procedure calls, and identifies problems with their knowledge. In particular, we show that even upper-level students suffer from difficulties that instructors might have assumed have long since been addressed.

CCS Concepts •**Software and its engineering →Procedures, functions and subroutines;**

Keywords Procedures, aliasing, call-by-value, call-by-reference, pedagogy

ACM Reference format:
Preston Tunnell Wilson, Shriram Krishnamurthi, and Kathi Fisler. 2017. Student Understanding of Aliasing and Procedure Calls. In *Proceedings of SPLASH-E, Vancouver, Canada, 2017,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction to Aliasing

Aliasing is a fundamental concept in programming: when do multiple different names resolve to the same entity? (This is analogous to aliases for humans: one person being referred to by more than one name.) Aliasing matters especially in languages with mutation operations, because when an entity is mutated using one alias, the other alias also "silently" changes; programs that assume otherwise will demonstrate errors that can sometimes be quite subtle. This can especially manifest in parallel programming errors: identifying too few aliases results in failing to lock shared objects, generating race conditions (which are notoriously difficult to find and debug); identifying too many results in excessive locking, reducing the performance benefits of multiple threads.

Aliasing is also important when studying program complexity. An aliased entity does not consume any extra space (other than that for the alias itself), which can reduce (in a sense measurable by complexity measures like $O(\cdot)$) a program's space needs. On the other hand, aliases can also keep objects alive for longer than expected, which can have a negative impact on memory management techniques such as reference counting and garbage collection. Therefore, knowing about aliasing is vital for reasoning about both the correctness and the performance of programs.

In what follows, we present a brief summary of aliasing. Readers familiar with the concept and with terms like variable and object aliasing may skip this portion.

SPLASH-E, Vancouver, Canada
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

In most programming languages, there are two broad kinds of aliasing: aliasing of *variables* and aliasing of *objects*. Variable aliasing is easy to see in C:

```
int q = 4;
void f(int p) {
  p = 3;
}
void g(int *p) {
  *p = 3;
}
int main() {
  printf("%d", q);  // prints 4
  f(q);
  printf("%d", q);  // prints 4
  g(&q);
  printf("%d", q);  // prints 3
}
```

The procedure `f` does not alias its parameter but `g` does. Therefore, a mutation *inside* `g` (to its local variable `p`) is visible to the caller, and the final `printf` displays `3` instead of `4`.

Many languages, such as Java, do not have variable aliasing. In contrast, most languages do have object aliasing, as this Java example shows:

```
class IntBox {
    public int b;
    IntBox(int i) { b = i; }
    void set(int n) { b = n; }
}
class Client {
  public static void main(String[] args) {
    IntBox b1 = new IntBox(10);
    IntBox b2 = b1;

    System.out.println(b1.b); // prints 10
    b2.set(5);
    System.out.println(b1.b); // prints 5
  }
}
```

Here, `b1` and `b2` are aliases to the same object. Thus, even though we invoke the `set` method only on `b2`, the value in the object bound to `b1` has also changed.

In both examples, we can see the essence of aliasing: though an operation is performed via one name, the change affects both associations. Nevertheless, the two kinds of aliasing are semantically very different. Variable aliasing is necessarily limited to the scope of a variable; when the scope of an alias ends, that alias disappears. In contrast, object aliasing is not tied to variables, since objects can reside indefinitely on the heap. Therefore, a programmer must understand the behavior of the whole program to measure the potential for object aliasing.

Contributions of This Paper In this paper, we develop on past studies (section 2) on student understanding of aliasing. We perform studies in two different courses at one university, one upper-level and one introductory, the former using Java and the latter a mostly-functional, education-oriented programming language called Pyret. Concretely, we make the following contributions:

1. In each course, we study student understanding of aliasing at different points in the semester to identify improvements (or their absence) across that course (section 4, section 5).
2. We explore the evolving terminology that students use to describe potential aliasing in programs (section 7).
3. We summarize specific inaccurate models that students invoke when describing behavior of our study programs; these suggest concept-inventory distractors (section 8).

In general, faculty would benefit from having simple instruments for evaluating their students' understanding of aliasing. To that end, our work is also a preliminary step towards creating a concept inventory (Hestenes et al. 1992) for aliasing. We note that the concept inventory-style of question—short programs with multiple-choice answers—lend themselves particularly well to use with in-class peer instruction and clickers, which are an increasingly popular and effective education model (Porter et al. 2016).

Terminology Note In this paper, we will use "procedure" to also cover "method". Though technically not identical concepts, they both involve parameter-passing, which is the only aspect we care about in this document. Similarly, we will use "object" even for languages (such as functional languages) that have only "values", not objects in the sense of object-oriented programming, because again the difference is not germane. Finally, when we want to ignore the difference between aliased objects and aliased variables, we will use the term "entity".

2 Related Work

Past projects have studied misconceptions about variables, assignment, and parameter passing (Kaczmarczyk et al. 2010; Ma et al. 2011). These typically use short CS1-style programs containing a sequence of assignment statements; students are asked to explain the program's behavior. These studies show that students generally have both incorrect and inconsistent models of these concepts—particularly the underlying model of how memory works—early in a CS curriculum. Several differences to our work include: a focus on pointers and memory representations rather than the semantic concept of aliasing; focusing on only one language or paradigm; not accounting for advanced students (who, we surprisingly find, also fare poorly); and not identifying some of the issues we raise through our narrative analysis. Sorva (Sorva 2007) shows that students have many models of how objects are created and stored in memory.

Our work is more directly inspired by a recent paper (Fisler et al. 2017). We have used their problem sets as a starting point because they address the shortcomings of other work discussed above. However, their work also fails to consider students at multiple levels of preparation; their answers have ambiguous interpretations that we have fixed; and their focus is different (by examining inter-language transfer as well as emphasizing the interaction with scope).

3 Study Context

The quantitative results in this paper rely on information from a total of four studies, presented to students as quizzes. All four were conducted at a selective, private US university, in two courses. Both courses were taught by the same instructor, who is one of the authors of this paper.

3.1 Student Populations and Study Design

One, which we call **CS1.5**, is an accelerated introductory class taken mostly by first-year college students with some prior computing background; it compresses most of a CS1–CS2 sequence into a semester. Many students in this class did not have a sophisticated programming background, so the course could not assume prerequisite knowledge in terms of languages or concepts. The course itself used Pyret (https://www.pyret.org/), a new functional and object-oriented language designed for programming instruction. All the study instruments in this class used Pyret, since it was the one language students were sure to know.

The other, which we call **CSPL**, is an upper-level programming languages course taken mostly by third- and fourth-year college students, with some second-years, master's, and PhD students. Most students in this class had a much more sophisticated programming background. All knew some Java, all had had at least a year of collegiate computer science, most had much more computer science than that and also had industrial experience in summer internships or full-time. The study instruments in this class used Java.

We administered two quizzes in **CSPL**. The first, given early in the semester, established a baseline of their knowledge (since all students had fairly extensive experience with mutation in imperative programming). After this, aliasing was taught explicitly in class. The second quiz was essentially a repeat of the first, at the end of the semester.

In **CS1.5**, we administered two quizzes. We could not assume students were comfortable with imperative programming at entry, so a baseline would not have been meaningful (and may have induced a sense of inadequacy in some students). We administered the first quiz (similar in content to that of **CSPL**) immediately after the course introduced mutation. We taught students a model for memory by going over equality. We covered both shallow (referential) and deep (value) equality. The second quiz, like in **CSPL**, was near the end of the course and essentially a repeat of the first.

3.2 Study Questions

Figure 1 shows Java versions of our quiz questions.[1] In the style of a concept inventory, students were given multiple choices that correspond to one correct answer and distractors for common misconceptions (identified from the past literature). Students were also asked to provide a written description of the program behavior resulting in that answer. We also gave students the choice of indicating that they did not know the answer and to explain why.

`Question1` helps identify whether students have an (incorrect) variable-aliasing semantics in mind for method calls. Observe that the mutation to `toReset` is to a purely local variable, and hence has no impact on `leo`'s fields. Thus the output is `"Leonard"`.

[1]We have slightly abbreviated some of the class names and string constants to make the code fit in the page. The full version will provide a link to the complete code.

```
1  class Question1 {
2      public static void F1(Employee toReset) {
3          toReset = new Employee("Betty", 30);
4      }
5
6      public static void main(String[] args) {
7          Employee leo = new Employee("Leonard", 20);
8          F1(leo);
9          System.out.println(leo.name);
10     }
11 }
```

```
1  class Question2 {
2      public static void F2(Employee e1, Employee e2) {
3          e1 = e2;
4          e1.pay = 12;
5      }
6
7      public static void main(String[] args) {
8          Employee abel = new Employee("Abel", 7);
9          Employee betty = new Employee("Betty", 10);
10         F2(abel, betty);
11         System.out.println(abel.pay + ", " + betty.pay);
12     }
13 }
```

```
1  class Question4 {
2      public static void F4(Employee e1) {
3          Manager bill = new Manager("Bill", e1);
4          bill.employee.pay = 2;
5      }
6
7      public static void main(String[] args) {
8          Employee peter = new Employee("Peter", 25);
9          Manager glinda = new Manager("Glinda", peter);
10         F4(peter);
11         System.out.println(glinda.employee.pay);
12     }
13 }
```

```
1  class Question5 {
2      public static void F5(Employee e1) {
3          Employee e = e1;
4          e.pay = 150;
5      }
6
7      public static void main(String[] args) {
8          Employee jack = new Employee("Jack", 100);
9          F5(jack);
10         System.out.println(jack.pay);
11     }
12 }
```

Figure 1. Java versions of quiz programs.

`Question2` tests for whether students expect variable-aliasing and how objects can be mutated when passed as parameters. The correct answer is `7, 12`. This is because on line 3, `e1` is changed to alias `e2`, so the `"Abel"` object cannot be affected by anything further in `F2`. The change to `.pay` in line 4 thus modifies the `"Betty"` object.

The first-round quiz contained a `Question3` that passed `Strings` as arguments. Students' answers to this question revealed confusion about whether Java `Strings` were base- or object-types, mutable, etc. Since this confusion distracted from our core focus on aliasing, we dropped this question from the rest of the study.

`Question4` nests objects and then uses enclosing objects to mutate fields of interior objects. As the `"bill"` object holds an alias to `e1`, the correct answer is `2`.

`Question5` is intentionally a variation on `Question1`. In it, we simply introduce an extra local variable to alias the parameter, and mutate that local variable instead, instead of directly modifying the parameter. We discuss the reason for this variant in more depth later (section 5). The correct answer is `150`.

4 Student Models of Aliasing in CSPL

Our primary goal is to explore students' understanding of aliasing. Each question had answer options corresponding to each of aliasing and non-aliasing. We were interested in not only whether students predicted the correct answers for the quiz questions, but also whether their written responses reflected an understanding of the underlying mechanisms of aliasing (even if they did not use that terminology).

In **CSPL**, the first quiz occurred at the start of the semester, before aliasing or the semantics of mutation were covered in the course. Students' responses therefore reflected their prior programming experience. Given **CSPL** students' prior knowledge of Java and experience with the language and other languages that behave similarly—such as Python—we might have expected them to have no difficulty with these questions, but their performance failed this expectation. The following table summarizes performance of students in **CSPL** across the first and final quizzes using the programming questions (the questions were shuffled and the objects changed across versions, but the questions were structurally identical). The cells report numbers of students (out of N=32, the number who took both quizzes). These correspond to Questions 1, 2, and 4 in fig. 1 (`Question5` was not in the first quiz).

Question	Got Better	Both Wrong	Got Worse	Both Correct
1: var. aliasing	7	3	3	19
2: obj. & var. aliasing	4	6	1	21
4: obj. aliasing & nest.	9	0	1	22

The number of students who had a question wrong on the first quiz is the total of the "Got Better" and "Both Wrong" cells. On each question, roughly 30% of the class had the wrong answer in the first quiz. Half of the class missed at least one of these questions. Only six students got both `Question1` and `Question2` wrong, while five students got both of these correct while missing `Question4`. Misunderstandings around aliasing are thus common in this population, despite their experience.

Several students had a consistent interpretation of variable aliasing or object aliasing. A selection of them answered one question correctly and one incorrectly. They are an interesting population since they might have assumed variable aliasing inconsistently, or they have a broken model of it. We now examine this population. In the first quiz, the most common wrong answer on each of `Question1` (9 out of 10 students) and `Question2` (6 out of 10 students) assumed variable aliasing. Four students got one of these questions right and one wrong. Interestingly, one of the four described contradictory models across the two written answers, one assuming leakage outside the function and one assumed containment of modifications within the function. One had answers too vague to suggest models, and another cited "object references are passed by value", drawing different conclusions in each question. The fourth gave incomparable explanations for both questions.

Inferring Models from Explanations

While the students' answers suggested certain assumptions about aliasing, the more interesting question is whether their written explanations reflected those same assumptions. We initially hoped to categorize students' models from their explanations, but this ultimately proved too difficult. The authors made several attempts at a coding rubric, and repeatedly failed to achieve inter-coder reliability in applying it.

As an example of what went wrong, consider the following rule that we tried to include in our rubric:

> *Mark the student as correctly aliasing an actual argument if the student mentions "pass by reference" in combination with "changing one affects the other."*

Part of the problem with a rule like this is students used terms like "reference" in myriad ways, each of which came with different nuances. In the answers to `Question2` alone, for example, students used phrases such as "pass by reference", "assigns by reference", "refers to", "both variables are references to", and "passes references by value". Phrases using the term "reference" appeared in explanations of both correct and incorrect answers. Ultimately, the authors could not agree on the interpretations of such phrases consistently enough to form a coding rubric.

Other challenges included vague descriptions, use of pseudo-technical vocabulary in an informal context, and hints of multiple models in the same sentence. As the number of these cases increased, we realized that free-form responses were not a reliable way of eliciting students' models of aliasing and parameter passing. At the very least, we needed to avoid relying on students' vague understandings of terminology.

The granularity and variety of decisions embodied in a language semantics or language model pose additional challenges. Experts know core models that include the details of memory layout (e.g., a stack and a heap, and the mappings of variable names on the stack to heap addresses). Students, however, are initially taught and learn models of languages based on syntactic constructs. For example, in the **CSPL** first quiz explanations, we see many students attributing their answers (whether correct or incorrect) to the semantics of parameter passing, overlooking issues of scope. Students also conflate or miss issues, e.g., stopping with a description of parameter passing without considering the behavior of mutation.

5 Student Models of Aliasing in CS1.5

The student performance in **CS1.5** is less interesting for the following reason. The first quiz was given immediately after students were introduced to mutation (as discussed in section 3.1), so students did quite well. For the end-of-semester quiz, similarly to **CSPL**, we dropped the equivalent of the problematic `Question3`. To further reduce work at a busy time of semester, we also dropped the equivalent of `Question1`, on which students had done well initially. The performance across the quizzes on the remaining two questions was as follows (N=42):

Question	Got Better	Both Wrong	Got Worse	Both Correct
2: obj. & var. aliasing	5	1	1	35
4: obj. aliasing & nest.	2	4	5	31

As we see, students generally did much better. There are many possible reasons, given the differences in population, background, etc., and we need significantly more studies to tease them apart. We have already detailed one, which is the immediacy of student preparation for the first quiz (which could have also affected the second administration). Another important one is the following.

As discussed, **CS1.5** used the Pyret language. Though the underlying semantic model of Pyret (for the parts tested in this paper) is identical to that of Java, there was one problem with directly translating the code. Naively, the equivalent of the method `F1` (from `Question1`) would appear (in Pyret, which has a Python-inspired syntax) to be:

```
fun F1(toReset):
  toReset := Employee("Betty", 30)
end
```

However, in Pyret, mutable variables must be declared explicitly. Due to other design decisions, this must instead be written as:

```
fun F1(toReset):
  var temp = toReset
  temp := Employee("Betty", 30)
end
```

Though it is not clear this was intended by the language designers, we conjectured that introducing this extra "indirection" reduces the likelihood of students thinking that a mutation inside the procedure has any impact on the caller. This is why we added `Question5` to the final quiz in **CSPL**. Performance on this question in **CSPL** resembled that on `Question1` and `Question2`: 6 students did not choose the correct answer, with 3 of those saying that they didn't know. Nearly all students who got it right explained that the new local variable was simply another reference to the passed-in object; 2 students who didn't know wondered whether the new variable created an alias or a copy of the object.

6 Trying to Identify Models Visually

After finding it hard to elicit students' models of aliasing from explanations on the first quiz of **CSPL**—as evinced by the difficulty in coding their written responses—we introduced a new quiz at an intermediate stage of the course. Instead of asking students to generate explanations, we asked them to pick one, and to avoid ignorance of or confusion over terminology, we used diagrams. Concretely, we gave a single problem similar to `Question1` in section 3.2, but this time asked students to select the diagram that best depicted the relationships between variable names, memory addresses, and objects at a particular point in the program's execution. Figure 2 shows the diagram options. These diagrams had not been presented in class, so we were relying on students' intuitive readings, rather than an agreed-upon semantics for the diagrams. What follows is a description of the program which the diagrams model and our intended meaning behind the diagrams.

In the problem, a variable `x` (bound to an object) is passed as the actual parameter to a method with formal parameter `y`. In this

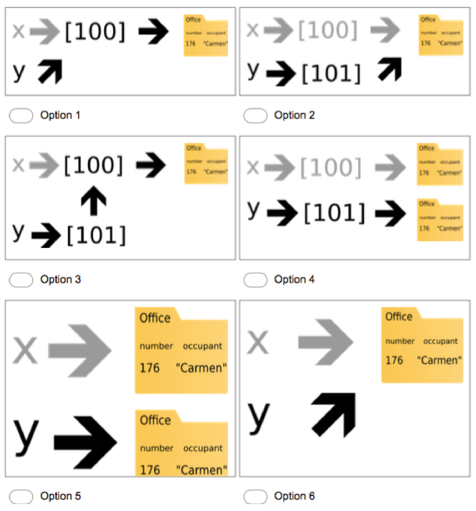

Figure 2. A visual presentation of possible models of variables, addresses, and objects. Students were asked what the state of the program looked like just after `x` (bound to an `Office` object) is passed as the actual parameter to a method with formal parameter `y`. Students were told that "grey parts represent aspects of the system which are 'out of scope,' 'unavailable,' or generally inaccessible." The larger size on the last two questions is not meaningful.

context, diagrams 4 and 5 are wrong because they have two copies of the object (the gold folder icon). Diagram 3 is wrong because it lacks a consistent type of mapping from names to objects (due to the extra reference from one address to another; in practice, a language with such a model could not guarantee constant-time access of objects via variable names).

Diagram 2 is the most correct: variables map to memory locations (the stack), which in turn map to objects (in the heap). One could make a case for diagram 6 if the variables were interpreted as references to objects (making the addresses implicit by conflating names and stack addresses). Diagram 1 suggests that two variables share the same memory address: this could suggest variable aliasing (if the address were on the stack), but it could also be correct if students interpreted the memory address to be that of the object in the heap (eliding the stack in the picture).

To be clear, we did not expect that the students necessarily knew about the separation of the stack and heap (though most would have had a prior course that covered this material). Mainly, we were curious whether their choice of diagram might shed light on the conceptual models they used to answer the questions on the first quiz. We also hoped the diagrams could provide a more reliable way to extract students' understanding of aliasing.

Because we did not teach students what these diagrams represented, we were relying on their intuitive readings, as we mention above. Alas, we hoped in vain. Table 1 shows which diagrams were chosen by students who correctly answered each question from the first quiz. The data show that roughly 25% of the students selected one of the incorrect visual models (#3–5), despite their earlier answers correctly reflecting that objects, but not variables, alias on procedure calls. Many students selected diagram 1, which

First Quiz Question	Share Addr [# 1] (n=12)	Stack & Heap [# 2] (n=12)	Chain Addrs [# 3] (n=3)	Two Objs [# 4/5] (n=6)	No Stack [# 6] (n=7)
1: var alias	7	10	3	5	4
2: obj. & var. alias	7	11	3	3	5
4: obj. alias & nest.	10	9	2	3	5

Table 1. Visual models selected by students answering questions correctly on the first-round quiz in **CSPL**. The "#N" annotations in column headings refer to diagrams from fig. 2.

enables the variable-aliasing interpretation. Only half of the students selected one of the viable diagrams (#2 and #6), despite having correct quiz answers. Of the 9 students who selected one of the incorrect visual models, four had a perfect score on the quiz, and another three missed only one question. Of the 5 students who got at most one question right on the first quiz, only one picked one of the incorrect visual models.

These results suggest that the visual models are not sufficient discriminators of misconceptions about the behavior of aliasing. As a result, we dropped the results of this survey from our analysis for **CSPL**. We still believe some diagrammatic notation could have value, but it clearly requires much more explicit instruction. We therefore leave this as an opportunity for study in future work.

Pedagogic Note On the other hand, pedagogically speaking, administering the diagram-based quiz was extremely helpful for in-class discussion. Since students had been forced to think about both the meaning of the pictures and what they thought was happening in the program, they came primed for a discussion of both. The visual vocabulary also made it possible for them to suggest alternate pictures to explore other models they had in mind, and to comment on proposals from each other. In other words, while these pictures are—in the absence of any prior explanation—a poor *diagnostic* device, they appear to be an excellent *pedagogic* one.

7 Evolution of Descriptions

One of our study goals was to track how student understanding of aliasing evolved over the two courses. Understanding of aliasing is evidenced in many ways, including changes in performance on quiz questions, more precise free-form explanations, and changes in use of vocabulary related to aliasing.

The table in section 4 shows the performance change in **CSPL**. Students who were still wrong in the final quiz are in the "Both Wrong" and "Got Worse" columns. The table shows that most students resolved confusion on `Question4`, but many students were still (or newly!) wrong on the first two questions (still-wrong students typically picked the same wrong answer both times).

In terms of vocabulary, none of the **CSPL** students used the term "alias" in the explanations in the first quiz (hardly surprising). Only 3 students used this term in the final quiz (and only one used it on all three questions). Thus, despite being introduced very explicitly in class, the aliasing vocabulary failed to take hold. Use of the phrase "call by value" dropped: 6 students used it in the first quiz, but only two used it in the final quiz, both in the form "Java passes references by value". Only one student (out of four) who had used the phrase "pass by reference" in the first quiz was still using it in the final quiz, though two students started using this terminology. Descriptions in terms of "references" remained: only six students never used a variant on "refer(ence)" in the final quiz: one wrote in terms of "alias", while 3 wrote in terms of "points to". It is worth noting that none of these terms suffice to help us (the researchers) determine which fig. 2 diagram the students might have in mind.

The vocabulary shift in **CS1.5** was more interesting. Half of the students (21 out of 42) used aliasing terminology in the first quiz, which makes sense because the quiz was administered just after the lecture on mutation (the same instructor as for **CSPL** again introduced the vocabulary of "aliasing" very explicitly). By the final quiz at the end of the term, however, only 8 students were still using that vocabulary. Some switched to using "points to", while others spoke in terms of "refer(ences)". This could be an artifact of prior programming experience (which many of these students had), or students could have simply forgotten the terminology from the mutation lecture. It would be interesting to re-quiz these students at the start of the next academic year to see if their quiz performance drops without the immediacy of the mutation lecture.

8 Apparent Student Models

Eventually, our goal is to get to a concept inventory on aliasing. Developing this requires a clear understanding of the misconceptions and incorrect models that students might hold. Here, we describe several models that appear in students' free-form explanations of their quiz answers. We present only the models that correspond to student confusion; most students did not exhibit these problems.

- *Assignment by Copy*: a few students in both **CSPL** and **CS1.5** mentioned an assignment by copy mechanic, probably similar to a "pass by copy" or "pass by value" semantics. Sample student quotes towards this model include:
  - *I'm not sure if bill holds a copy of e1 or an alias*
  - *I can think of several things that "o2 = o1" might do. Let's pretend that it doesn't change o1, and makes a copy of o1 for 'o2' to henceforth refer to.*
- *Reference Grouping*: One of the **CS1.5** students described a model of memory in which aliases get grouped together. The following quote is about a program that attempts to perform a "swap" of parameters via local variables (which was on the first quiz but dropped for the final quiz).

  temp, o1, and occupant1 all refer to "Carmen" at first, while o2 refers to "RAs" at first. Then, o1's value is changed to o2, so all four of temp, o1, occupant1, and o2 refer to "RAs". Finally, o2's value is changed to temp, which doesn't do anything since they're all the same.
- *Bidirectional assignment*: Following an assignment statement of the form `o1 = o2`, where `o2` is a parameter, some students believe modifications to `o1` reflect in `o2`. This assumption can get conflated with whether updates leak through to actual parameters.

We note that the last two interpretations are more consistent with languages whose behavior is based on unification—such as logic programming languages (with Prolog as a leading example).

9 Conclusion

Our paper focuses on aliasing, a key concept in understanding programs for both correctness and performance. Our studies show confusion about aliasing across programming languages and even in students with significant education and experience. We find that students have difficulty articulating their mental models. We have also taken steps towards creating concept inventory-style questions for aliasing, which are useful in both education and assessment.

In the future, we believe it would be worth teasing out some of the misconceptions we find to study in greater depth. We also believe there is value to trying to formalize and explicitly teach the visual models we have used, and studying whether their use has an impact on student understanding. Finally, we wonder whether the standard procedure call terminology is unhelpful in clarifying the nature of aliasing, by failing to align the aliasing aspect of procedure calls with the aliasing of non-procedure-calls (such as assignment).

Acknowledgments

We thank Joe Gibbs Politz for alerting us to the connection to clickers, and appreciate the graduate students who helped us debug early iterations of the quizzes. This work is partially supported by the US National Science Foundation. The first author's last name is *Tunnell Wilson* (i.e., indexed under "T").

References

Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *ACM Technical Symposium on Computer Science Education.*

D. Hestenes, M. Wells, and G. Swackhamer. 1992. Force concept inventory. *The Physics Teacher* 30 (1992).

Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *SIGCSE.*

L. Ma, J. Ferguson, M. Roper, and M. Wood. 2011. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education* 21, 1 (2011).

Leo Porter, Dennis Bouvier, Quintin Cutts, Scott Grissom, Cynthia Lee, Robert McCartney, Daniel Zingaro, and Beth Simon. 2016. A Multi-institutional Study of Peer Instruction in Introductory Computing. In *SIGCSE.*

Juha Sorva. 2007. Students' Understandings of Storing Objects. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 127–135. http://dl.acm.org/citation.cfm?id=2449323.2449337