Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates

Kathi Fisler
WPI
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Preston Tunnell Wilson
Brown University
ptwilson@brown.edu

ABSTRACT

Scope, aliasing, mutation, and parameter passing are fundamental programming concepts that interact in subtle ways, especially in complex programs. Research has shown that students have substantial misconceptions on these topics. But this research has been done largely in CS1 courses, when students' programming experience is limited and problems are necessarily simple. What happens later in the curriculum? Does more programming experience iron out these misconceptions naturally, or are interventions required?

This paper explores students' understanding of these topics in the context of a programming languages class for third- and fourth-year CS majors. Our pre- and post-tests pose questions in two programming languages to gauge whether upper-level students transfer knowledge between languages. Many students held misconceptions about these concepts at the start of the course. Students made progress in only some languages and topics, and cross-language transfer does not occur naturally. We also discuss various pedagogic activities we used to engage students with these concepts, and provide data and student opinion on their effectiveness.

Keywords: scope; mutation; programming; concept inventories; notional machines

1. INTRODUCTION

Programming beyond the beginner level draws on core concepts such as scope, aliasing, and their interactions under mutation and parallelism. Because these topics are complicated and have many subtle interactions, students learn about them incrementally across several years.

Past studies conducted at the CS1 level (discussed in § 2) already demonstrate that students have considerable difficulties with these concepts in isolation. In this paper we study these questions with third- and fourth-year college students; by this time, their coursework, internships, and projects should have greatly improved their knowledge. In addition to (and because we are) considering the upper-level curriculum, we have four other novel aspects to our study:

- We study the *interaction* between the key features of scope, mutation, and aliasing.
- We conduct a study across two semantically very similar but syntactically different languages, to measure transfer across languages.
- We examine the impact of multiple forms of both *teaching* and *assessment* of these features.
- We include more features than in prior studies, even briefly considering closures (which are now a part of most mainstream languages, including Python, Java, etc.).

Concretely, we present the design of a multi-language quiz on scope, mutation, and aliasing. We administer this quiz as a pre- and post-test in an undergraduate programming languages course, and examine how the course impacted students' understanding. The two languages used in the quiz are Java and Scheme, which have identical semantics [4] despite very different syntaxes and surface-level concepts.

Our findings, which we discuss in detail, are as follows:

- Despite their advanced status in their education, students did poorly on the pre-test, even in Java, which they had used in several courses and in software engineering projects. The results here are similar to those from CS1 courses (elsewhere), despite the students' having two to three more years of programming experience.
- Students improved significantly by the end of the course.
- Most of the improvement was in Scheme, not in Java. Though the course did program in a variant of Scheme called Racket, it did *not* itself use the features being tested—rather, it showed *how to implement them*. Therefore, it raises the question of whether seeing how to implement a language feature might be more useful than having experience with it in limited settings.
- The improvements were not uniform across features, either. There was significant improvement in the understanding of scope and of variable mutation, but only borderline improvement on questions involving mutation of function/method parameters.
- The changes in performance did not correlate well with the different kinds of teaching and assessment. Nevertheless, in surveys, almost all students reported finding these techniques at least somewhat useful.

2. RELATED WORK

Our quiz is a first step towards a *concept inventory* for scope, mutation, aliasing and their interactions. A concept

SIGCSE '17, March 8–11, 2017, Seattle, WA, USA

ISBN 978-1-4503-4698-6/17/03…$15.00

DOI: http://dx.doi.org/10.1145/3017680.3017777

inventory is a collection of questions that test understanding of how a particular concept works; the Force Concept Inventory in Physics [2] is the classic example; similar efforts are underway for a few areas in CS [7]. An inventory is typically a set of multiple-choice questions in which each option has been validated to indicate a specific incorrect conceptual model; these incorrect models are called *misconceptions*. The validation process is time-consuming and typically requires several iterations. The quiz we discuss in this paper has not yet been validated, as we are in the first phase of identifying questions on which upper-level students perform poorly. The incorrect answers in our quiz are based on two decades of teaching this material and observing certain consistent patterns of mistakes that students make, and therefore offer a useful starting point towards eventually arriving at a concept inventory.

Several projects have studied misconceptions about variables and parameter passing [7, 9]. 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. Futhermore, Sorva [13] showed that students have many models of how objects are created and stored in memory. Memory models are necessary but not sufficient in our work, which also studies scope (and its interaction with mutation). Fleury [5] found that many students assumed dynamic scope when confronting unbound variables in Pascal. Our questions also check whether students use dynamic scope when there are multiple bindings to the same name. We have not found prior studies that target the interaction of scope and mutation with an emphasis on nested scopes.

Chi [1] suggests that misconceptions, once identified, cannot simply be replaced with correct models. Cognitively, misconceptions need to get layered over with new information that is retrieved more often than prior models. The nature and extent of learning to layer over misconceptions is an open question [6, 12], particularly in the context of computer science. We report on four instructional techniques that we used to help students refine their understanding of scope, mutation, and aliasing, though we do not attempt to measure (beyond student opinions) which is most effective.

Students' models of program behavior are closely tied to notional machines [3], which are abstract models of how a system (such as a specific programming language) should behave. Notional machines can under-approximate the full semantic behavior of a language, giving students a partial conception of constructs that is consistent with what they have learned so far. We conducted our study in a programming languages course in which students wrote interpreters to explore the workings of certain constructs. In writing interpreters, students had to explicitly write notional machines in code. It is not clear, however, whether the act of writing these machines helps students understand the results of running them. That question is among those we are exploring in this work.

3. BACKGROUND: SCOPE AND ALIASING

We start by explaining the key concepts of scope, aliasing, variable mutation, and object mutation, on which our quiz tests students.

Variables and Aliasing.

Consider the following Java program:

```
void m(C o) {
  int x = 3;
  int y = x;
  x = 4;
  o.f = 5;
}
```

The method `m` takes an object `o` (of type `C`) as a parameter. It defines local variables `x` and `y`. There are then two assignment statements: one updates the value of `x`, while the second changes a field (`f`) of object `o`. We call the first *variable mutation* and the second *object mutation*.

These two forms of mutation are quite different, even though they appear similar syntactically. In most languages, variables do not *alias*. That is, changing `x` above changes the value of `x` only, and does not alter the value associated with `y`. In Java and Scheme (the languages in this study), function calls do, however, alias objects. In this example, suppose we run `m(p)`, where `p` is some object of type `C`. Once the method completes, the value of `p.f` will also be `5`, even though the program contained no statement of the form `p.f = ...`. This is because `o` *aliases* `p`.

Aliasing is a vexing problem in complex programs. It is especially thorny in parallel and concurrent programming, where aliases greatly expand the sets of readers and writers (and can therefore hurt thread-safety). For this reason, enormous research effort has gone into reining in aliases ([10] is an illustrative example); industrial languages like Mozilla's Rust now provide "ownership" annotations for this purpose. Therefore, it is important to gradually convey this important idea to students, due to its impact on the understandability, efficiency, and parallel behavior of their programs.

Binding and Scope.

Scoping rules determine the *extent* to which a binding to a variable name stays in effect, as well as which one is used when multiple variables share a name. Consider the following program in a hypothetical language:

```
def f():
  x = 3
  g(x)
def g(y):
  print(y + x)
```

Assuming `g` was called from `f`, what happens at the `print` line? Does `x` still have the value `3`—which corresponds to *dynamic* scope—or is it an error because `x` is not bound (corresponding to *static* scope)? Most languages have chosen the latter interpretation; indeed, even some languages that began with dynamic scope (Lisp, JavaScript, etc.) in some situations have switched to static scope. Yet in our experience, some students are truly confused while many assume the language will have dynamic scope.

Scope can become especially significant when programming with closures (a.k.a., "anonymous functions", "delegates", etc.). While closures may have seemed idiomatic only of functional programming, they are now widely found in all manner of languages. For instance, they are in JavaScript, Scala, Python, and are now even a part of the standard library interfaces in Java 8. Therefore, understanding the interaction of scope and aliases, and both of them with mutation, is increasingly important in all the families of languages that students will encounter.

Sidestepped Subtleties.

Simply in the interaction of scope, mutation, and aliasing, there are many more issues we could consider. For instance, languages like JavaScript and Python have complex "lifting" rules for variable declarations that are not provided at the beginnings of blocks. In Python, many scope rules are subtle, and certain nested declarations are surprising, essentially breaking the expectation—established since Algol 60—that static scope is *lexical* [11, § 4]. In some languages, attempting to set a variable's or field's value before defining it results in an error, but in many "scripting" languages, this usually just introduces the variable or field; in some of them, even accessing a missing field does not signal an error. We consider all these and other such issues outside the scope of our study, though we can certainly extend this work to cover more of these topics, and certain instructors who do rely on such language features should consider doing so.

4. A QUIZ ON SCOPE AND ALIASING

For this project, we focused on the behavior of scope and aliasing (exercising them through mutation and parameter passing) in two languages with similar semantics for these topics: Java and Scheme. Working with two languages lets us assess whether students understand these issues conceptually, beyond the context of a particular language. We chose Scheme and Java because our student population had been exposed to both during their coursework. Many other languages share the Java/Scheme semantics of scope and aliasing, so our quiz can be adapted to other languages.

Our quiz targets understanding of the following concepts:

- If there are multiple bindings (scopes) for the same variable name, which one is referenced at a given use? Which are affected by an assignment to that name? These cover static, dynamic, and nested scopes.
- When a variable name is used as an actual parameter, are the variable and the parameter aliases?
- When an object is passed as an actual parameter, does the parameter reference the passed object or a copy of it?
- When one local variable is assigned to another, are they aliases? Does changing one affect the other?

Concretely, our quiz contained 18 multiple-choice questions: 12 in Scheme and 6 in Java. (We used more Scheme examples because they are far more concise than those in Java.) Each question asked for the result of running a short (4-10 line) program, giving 4-6 answer choices. Figure 1 shows two examples.[1] The given choices captured each interpretation or misconception we could identify through the literature, our experience, and brainstorming. In some cases, we added implausible answers to check whether students were merely guessing instead of taking the quiz seriously.

5. STUDY LOGISTICS

We tested our quiz in a programming languages course for third- and fourth-year undergraduate CS majors at WPI. Students used Scheme for programming activities throughout the course, following the "definitional interpreter" model used by several books, including the course text [8]. The vast majority of students had used Scheme in their introductory course (though that knowledge was rusty at the start of this course); all had Java experience from one or more of the CS2 course, the software engineering course, and internships.

[1]The details are on the Web at `cs.brown.edu/research/plt/dl/sigcse2017/`.

Question Set (size)	Pre	Post	p-value
Scheme (12)	66%	88%	2.529402e-09
Java (6)	65%	69%	0.1452405
Scope (7)	66%	86%	2.536865e-06
Param Mutation (5)	62%	70%	0.05071296
Var Mutation (7)	67%	85%	3.198021e-06

Table 1: Class averages (N=66) on subsets of quiz questions. Significance of the gains was calculated using a Wilcoxon signed-rank test.

We gave the quiz during the first week of the course (before discussing either scope or mutation), then again after all lectures and activities on scope, mutation, and parameter passing had been completed. Students took the quiz online outside of class both times. Students had received automated grading results on all related activities before they took the post-quiz. The questions on the post quiz were identical to those on the pre quiz, but were presented in a different order. In both the pre- and post-tests, the Scheme questions appeared before the Java questions.

For both the pre- and post-tests, students were told that the quiz was a diagnostic that the instructor was using to check whether the activities were effective at helping them learn the material. Failure to take the quiz at all would count against their course grade, but incorrect answers would not. These statements were designed to incentivize participation without cheating (by running the programs and entering the correct answer, which the number of incorrect answers suggests students did not do). Students almost never chose answers that were implausible, suggesting that students were not randomly guessing either.

6. RESULTS AND OBSERVATIONS

6.1 Comparing Pre and Post

Table 1 summarizes overall class performance on the pre- and post-quizzes. The table reports on five clusters of questions: questions on Scheme alone, on Java alone, on scope, on mutation of (fields of) parameters, and on mutation of local (non-parameter) variables. Each question on the quiz is covered in two rows, one based on language and one based on topic; one question is counted in both the scope and variable mutation topics.

The quizzes suggest that students start out with only middling knowledge of both languages and better command of Scheme than Java afterwards. This is somewhat surprising, since nearly all students had more experience in, comfort with, and preference for Java. The least performance improvement occurs on the topic of parameter mutation, a topic tested more heavily in Java (4 questions) than in Scheme (1 question). In contrast, the quiz used Scheme more to test scope (5 questions versus 2 in Java) or variable mutation (6 questions versus 1 in Java). We cannot tell from this data whether the key factor in student performance was lack of understanding of the topic, greater facility in one of the two languages, or failure of students to transfer what they were learning about these constructs from Scheme to Java (something we discussed during lecture, but did not exercise explicitly in homework).

```
(let ([x 3])
  (let ([f (lambda (y) (+ x y))])
    (let ([x 5])
      (f 10))))
```

Choices: 13 (correct), 15, 18, error

```
class Question {
  static int a = 3;

  public static void F5(int b) {
    int a = 7;
    a = b;
  }

  public static void main(String[] args) {
    int v = 9;
    F5(v);
    System.out.println(a + v);
  }
}
```

Choices: 10, 12 (correct), 18, error

Figure 1: Sample questions in Scheme (L) and Java (R), both of which distinguish static and dynamic scope.

At the level of individual questions, there were only 3 questions on which at least 80% of the class was correct in both the pre- and post-quizzes: these tested basic nested scopes (Scheme), lack of aliasing between local variables (Scheme), and aliasing of objects passed as parameters (Java). No other question had more than 70% of the class correct on both pre and post. The question with the worst performance on the post-test explored whether assignments to parameter names were visible in the calling context (Java): over a third got this question wrong both pre and post, and 12% *lost ground* on this question between the two quizzes; only a third were correct both pre and post. Consistent with table 1, the number of students improving on each question was higher for Scheme questions than for Java ones. Losses were also more frequent on the Java questions (though losses occurred on every question sans one).

These data tell us that the quiz identified concepts that many students did not understand well, despite being upper-level and having significant programming experience (particularly in Java). The data also suggest that students did not learn to apply what they were (supposedly) learning in Scheme to understanding Java, despite the languages having a common semantics and this being discussed in class. To better understand these trends, we need to look at whether students had consistent misconceptions about concepts, and how often students used different interpretations of the same constructs across Scheme and Java.

6.2 Are Students' Models Consistent?

Some core topics (such as static scope) and potential misconceptions (dynamic scope; aliasing of identifiers passed as parameters) featured in multiple questions. For semantics topics common to multiple questions, we checked whether students chose consistently. Students grew more consistent (though not perfectly so) in the post-quiz on both static scope and whether variables aliased parameters. Figure 2 shows the histograms of how often students chose a static scope interpretation in each of pre and post.

Consistency Across Languages.

The quiz also contained pairs of questions that tested similar (though not identical) uses of the same concept in each language. For these pairs, we checked how often the correctness of students' answers correlated across the languages. Table 2 summarizes relative correctness across languages for

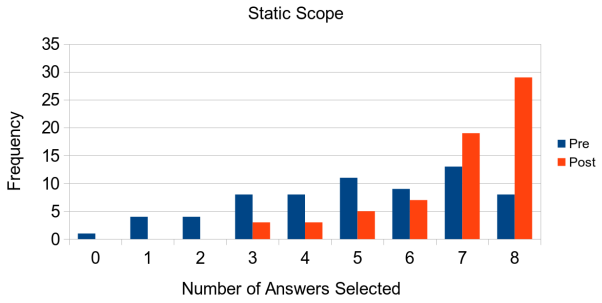

Figure 2: Distribution showing how many times students selected the static-scope interpretation during the pre- and post-tests. Students leaned more towards static scope (always the correct answer) in the post-test.

three question pairs: one on static vs dynamic scope, one on whether variables are aliased to parameters, and one on the effects of modifying a variable defined in an external scope.

In all three cases, a significant number of students answer the questions inconsistently both pre and post (the ✓× and ×✓columns). The balance between those columns skews in the post-quiz, however, with many fewer students being incorrect in Scheme and correct in Java. For the latter two questions, there is no correlation between the languages in the pre-quiz, but correlation emerges in the post-quiz. In general, improvement in these three pairs show stronger gains in Scheme than in Java in these common concepts. At least a third of the class remains inconsistent across languages in the post-quiz on questions involving variables.

It is important to eliminate one possible explanation for these changes. One might assume that students were using these features heavily in Scheme in the course, and therefore learned through experience how they behave. However, they did not: the style of interpreters written in the course does not exercise these features (mainly because the interpreters themselves do not use any mutation). Therefore, they could not have gained familiarity just from use.

7. INTERVENING ACTIVITIES

Between the tests, students did several activities designed to help them understand the quiz topics. Specifically:

	pre				post			
Topic	✓✓	✓×	×✓	χ^2	✓✓	✓×	×✓	χ^2
Scope	28	5	19	.007	47	16	1	.0006
Alias	19	17	14	.71	28	22	5	.002
XMut	21	17	16	1	40	18	4	.005

Table 2: Performance on similar concepts across languages. "Scope" contrasted static and dynamic scope, "Alias" checked leakage of assignment to a parameter in the calling context, "XMut" checked mutation of a variable defined in an enclosing scope. Column headings capture performance on Scheme then Java: ✓× means correct in Scheme and incorrect in Java. Cells contain counts of students (N = 66). Correlations computed with McNemar's test.

- They wrote an interpreter to implement static scope in a language with higher-order functions (i.e., closures as values) but no mutation. Along with the interpreter, they submitted a suite of tests designed to capture the intended scoping behavior in this language.
- They wrote a test suite to capture the behavior of a language with static scope, variable mutation, simple object-like data with field mutation, and parameter passing that aliased objects but not variables.
- They peer-reviewed each other's test suites from the previous two assignments. Each student reviewed two anonymized submissions. Reviewers were asked to comment on both the correctness and thoroughness of the submitted tests. Reviews were spot-graded, and a student could lose points for missing straightforward flaws in reviewed work.
- They answered clicker questions during lecture about the behavior of programs with these features. Participation in clicker polls counted towards their grade, but they did not lose points for incorrect answers.

Interpreters were graded on correctness against a test suite of programs written by the course staff. Test suites were graded for two metrics: *correctness* relative to a reference implementation (written by course staff) and *thoroughness* relative to errors that the test suite could detect. For the latter, the course staff wrote roughly 20 incorrect implementations of the corresponding language. Incorrect implementations would change the semantics in various ways (dropping variables from scope, adding them to inappropriate scopes, failing to update memory upon mutation, etc). A test suite received points for each broken implementation on which some test produced the wrong answer against the broken implementation (we call these implementations *coals*).

We used this combination of language implementation, testing, clickers, and peer review because they embody different ways of thinking about the concepts. Each promised potentials and pitfalls for student learning. Writing interpreters requires students to articulate a notional machine in code, but that alone doesn't require students to think through the consequences of running that machine. Writing tests requires students to articulate concrete examples of how a program should behave; this has the potential to help students think through interactions between constructs, but insights are limited to the programs that a student thinks to test. Peer reviewing can expose students to examples beyond what they thought to test, but random assignment of reviewers might not give students sufficiently different work to review. In addition, students might not reflect sufficiently on the tests they are reviewing to realize subtle points raised by those examples. Clickers give quick and immediate feedback, but the examples we used were very short (like the quiz questions) so as not to take too much class time.

Influence of Interventions

Our data are not rich enough to indicate which of our instructional activities (if any!) led to performance gains on the post-test. To measure that, we would have needed to assess students' understanding after they submitted each assignment and after they performed peer review on each assignment. We opted for a lighter-weight, single post-test during this first study. While this does not suggest which activities may have *caused* improvements, we can look at performance indicators on these assignments that *correlate* with improvements on subsets of questions, as well as students' perceptions of the activities.

Relating Coals to Quizzes.

For each of the scope and variable mutation topics, we checked whether changes in quiz performance (pre to post) correlated with whether students had test cases that covered these topics. Concretely, we used Kruskal-Wallis tests to check whether detecting each coal that broke static scoping (the expected behavior as stated on the assignment) correlated with whether students gained or lost ground on the scope questions within the quiz. We performed similar checks on questions that broke the semantics of mutation.

We found a strong correlation between the quiz question on dynamic scope and testing for the coal that implemented dynamic scope ($p = 0.00956$). A similar correlation ($p = 0.00791$) arose between the dynamic scope coal and another question about scope in the context of closures. No other combinations yielded correlations below $p = .05$.

On mutation, we found only one weak correlation ($p = 0.0474$) between a Scheme question on aliasing (whether the value of y changes after $y := x$ and $x := 5$) and a coal in which mutation affected the stack rather than the heap.

On the whole, whether students gained or lost ground on the quizzes does not correlate with the thoroughness of their test suites. Students knew their tests were graded for thoroughness, so they were incentivized to test carefully. That we found no correlation suggests that students can have a good understanding of the concepts without being able to articulate interesting examples of them, or vice-versa. Interestingly, we will see that students nevertheless felt writing test cases helped their understanding of the material.

Relating Interpreters to Quizzes.

Writing an interpreter corresponds to implementing a notional machine in code. This might force students to understand the semantics of constructs (and in turn help with the quiz). This view underlies the design of many programming languages courses that emphasize writing interpreters. Yet in the post-quiz, 9 (of 66) students still did poorly on the scope questions, 5 of whom had high grades on the interpreter. This is a small percentage, but still noticeable. The extent to which writing interpreters helps students understand language semantics, especially in situations that force multiple constructs to interact, remains an open question.

	Writing Interpreters				
Writing Tests	Much worse	Bit worse	Bit better	Much better	Total
Much worse	-	-	-	1	1
Bit worse	-	1	1	-	2
Bit better	-	-	17	15	32
Much better	-	-	10	28	38
Total	-	1	28	44	73

Table 3: Students' perceived impact of writing tests and interpreters on their understanding.

Students' Impressions.

The post-quiz asked students several Likert questions about the impact of our instructional activities on their learning. Each question took the form of "How has X impacted your understanding of how scope, mutation, and parameter passing behave?", where there was a question with each of "writing test cases", "writing or understanding interpreters", and "peer review (writing or receiving)" in place of X. The answer options were "Strong positive impact (I understand the topic much better having done this)", "Some positive impact (I understand the topic a bit better having done this)", and corresponding versions for some and strong negative impact.

Table 3 contrasts students' opinions of interpreters and tests. The data show students fairly evenly split as to which of the two was most impactful (contrasting the "strong positive" and "some positive" cells). Very few students felt an adverse effect from either activity. All but 4 students reported that peer review was a positive factor. We also asked how often clicker questions had helped to clarify material: 34 students selected "once or twice", while another 26 felt clickers helped "three or more times".

Overall, these data suggest that each of our activities has a potential role to play in helping students master these concepts. Further studies are required to understand the conditions that make each of these activities effective in practice.

8. DISCUSSION AND FUTURE WORK

Scope and aliasing are subtle concepts that programmers confront throughout their careers in various contexts. Educators cannot expect students to develop a solid understanding of these ideas and their interactions in the first year curriculum alone. We need pedagogic techniques and assessments that grow and evolve understanding of these topics over an entire curriculum.

Students' poor performance on Java questions in our pretest shows that extended experience programming in a language does not suffice to develop expertise in its semantics. Students continue to need explicit instruction to build accurate models of program behavior. This seems particularly true of questions involving mutation, given the low scores on this topic in the post-test. That our students progressed more on Scheme questions than on Java questions in the post-test suggests that upper-level students do not readily transfer knowledge gained in one language to another, even when that transfer is raised during lectures. The nature and extent of instruction needed to enable cross-language transfer in upper-level students is an interesting open question.

The relative pedagogic merits of our four activities (writing interpreters, writing tests, peer review, and clickers) need further study. Students report significant impact of each of these activities on their learning, but these impacts are not evident in our data. We believe that writing tests, for example, offers a valuable window into students' understanding of the space of problems around a language construct. Developing assessments that correlate students' testing behaviors with some other measure of conceptual understanding would help unearth the pedagogic connections.

Although we did our study in a programming languages course, the issues we raise are not limited to such courses. Indeed, any student who writes moderately complex programs—much less ones involving parallelism, concurrency, or distributed transactions—will confront the issues in this paper.

Doing analysis for this paper revealed that our quiz needs to rebalance the numbers of questions about specific topics across the languages; we need more Scheme questions that match some of the Java aliasing ones, and we need a few more questions overall to try to identify students' conceptual models of aliasing and its interaction with scope. We will revise the instrument for future iterations of this study.

Acknowledgments and Notes.

We thank the US NSF for its support. The third author's last name is "Tunnell Wilson" (index under "T").

9. REFERENCES

[1] M. Chi. Common sense conceptions of emergent processes: Why some misconceptions are robust. *Journal of the Learning Sciences*, 14:161–199, 2005.

[2] H. D., W. M., and S. G. Force concept inventory. *The Physics Teacher*, 30, 1992.

[3] B. du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*, 51(2):265–277, 1999.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, 1998.

[5] A. E. Fleury. Parameter passing: the rules the students construct. In *SIGCSE*, 1991.

[6] A. Gupta, D. Hammer, and E. Redish. The case for dynamic models of learners' ontologies in physics. *Journal of the Learning Sciences*, 19:285–321, 2010.

[7] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *SIGCSE*, 2010.

[8] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. 2006.

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

[10] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming*, 1998.

[11] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The Full Monty: A tested semantics for the Python programming language. In *OOPSLA*, 2013.

[12] J. Slotta and M. Chi. The impact of ontology training on conceptual change: Helping students understand the challenging topics in science. *Cognition and Instruction*, 24:261–289, 2006.

[13] J. Sorva. Students' understandings of storing objects. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, Koli Calling '07, pages 127–135, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.