Teaching Statement

Daniel R. Licata

I love teaching, and have successfully taught undergraduates at Carnegie Mellon and Brown. I have

- Lectured to small (20-30 students), medium (50-80 students), and large (150-225 students) classrooms, for 2 semesters for each size; and taught interactive lab sections for 9 semesters.
- Designed a course from scratch, including the course's content, activities, and assessments; and refined the course design during three semesters teaching it.
- Integrated research ideas into a course for first- and second-year undergraduates.
- Managed a staff of 15-18 teaching assistants (TAs). In two of three semesters, one of my grad TAs won the department teaching award. Also, I instituted an undergrad TA system for the course, where students just out of the course become TAs, and then gradually take on more responsibility over time.
- Instructed both computer science majors and non-majors, in both very tech-focused (Carnegie Mellon) and liberal arts (Brown) environments.

Much of this experience was after getting my PhD, when I designed and delivered a new introductory computer science course on functional programming, parallelism, and software verification (*15-150: Functional Programming*[1]) for three semesters at Carnegie Mellon University (CMU). The audience consisted of 80 students in Spring 2011 (all computer science majors and first-years); 205 students in Fall 2011 (including majors and many non-majors, mostly sophomores and juniors); and 225 students in Spring 2012 (both majors and non-majors; mostly first-years and sophomores). I designed the course content, the pedagogical activities (lectures, labs, homeworks), and the assessments, with the help of my PhD advisor Robert Harper during the first semester, and some excellent TAs throughout. I gave all the lectures all three semesters, instructed students during lab sessions, and managed the TA-student interactions. Students performed well on a series of difficult homeworks and projects. All computer science majors go on to take a required Parallel Data Structures and Algorithms course (taught by theory faculty), which builds directly on the skills introduced in my course, and the instructors of this course were happy with my students' preparation. Additionally, students rated me as excellent on course evaluations, and strongly praised my teaching in written comments.

Before that, I was a TA for 8 semesters. As a grad student at CMU, I TAed our undergraduate programming language class (sophomores and juniors; around 50 students), and our undergraduate constructive logic class (juniors and seniors; around 30 students); both had weekly lecture-style recitations with 20-30 students. At Brown, I TAed our two-course intro CS sequence, which taught functional and object-oriented/imperative programming, for six semesters.

In the rest of this statement, I will (1) describe how I integrated research-level ideas on parallelism into my introductory course, (2) describe an aspect of my teaching philosophy and how it influenced my practice, and (3) summarize my course evaluations.

[1]Full course materials are available at `www.cs.cmu.edu/~15150/previous-semesters/spring-2012`

Parallelism in the Intro Sequence

Recently, the CMU CS department redesigned its introductory sequence, with the fairly ambitious goal of integrating research ideas on *deterministic parallelism* by Guy Blelloch and others into first- and second-year courses. The objective is to teach students to see the inherent opportunities for parallelism in problems. My course introduces these ideas, which are developed further the following semester. I was aware of this research, but had not done any work on it myself, and was largely learning it as I taught it. The following examples illustrate how I made these ideas accessible and compelling to young students.

I introduce deterministic parallelism in the first few minutes of the first lecture: The students are arranged into rows in a large classroom. I say that, first of all, I want to get a little background information: how many of them have programmed before? They start to raise their hands, but I tell them to put them down. Instead, we're going to count how many people have programmed before as follows: We start from the first student in the top-left corner of the room, who knows whether or not she has programmed before. She passes this number to the person next to her, who adds one if he's programmed before, and otherwise just passes it on. This takes a little while for the top row. Then we do the same thing for the next row. But I don't really want to spend the whole lecture counting up how many people have programmed before, so let's try this instead: *In parallel*, let's count up each row, and then we can add up these counts to get the final answer! After acting this out, we write the code. This little counting exercise illustrates the main ideas of deterministic parallelism: The computation has a well-defined answer (how many students have programmed before?), and parallelism is just a means of getting to that answer more quickly. Moreover, the parallelism arises naturally from (the lack of) data dependencies—-the count for one row doesn't depend on the count for another, so we can do them in parallel.

The next example shows that parallelism fits in naturally with standard examples in a programming course, such as sorting. After implementing the usual mergesort on lists, we analyze its asymptotic time complexity, including both the total amount of *work* done, which is the usual sequential complexity, and the *span*, which is the length of the longest data dependency. The span provides a bound on the the running time with unlimited computational resources, and there is a theorem relating the work and span to the running time with a fixed number of processors. You might think that, because mergesort does a tree-based decomposition of problems (to sort $[6, 4, 5, 3, 2, 7, 8, 1]$, sort $[6, 4, 5, 3]$ and $[2, 7, 8, 1]$, and therefore $[6, 4]$ and $[5, 3]$, etc.), the span would be $O(\log n)$. However, mergesort requires auxiliary operations for spliting and merging lists, and on lists these are inherently sequential, and take linear time even with unbounded computational resources—so the span is $O(n)$. We conclude that lists are a bad data structure for parallelism, and start to study trees, which naturally support binary parallelism. Mergesort on trees is a more complex piece of code, but it is an appropriate increment in difficulty for this point in the course.

The next example shows that students at this level can do compelling projects with deterministic parallelism. After the midterm, we introduce a new data structure called sequences, which support n-way vector parallelism, and students implement the Barnes Hut algorithm for simulating the motion of celestial bodies using Newton's laws of motion and universal gravitation. In lecture, we cover the naïve algorithm for n bodies, which computes all n^2 mutual forces, and can be easily implemented using sequences with only $O(\log n)$ span (in parallel, compute the force on each body due to all other bodies, and, for each body, add the n forces due to the other bodies on a tree). Barnes Hut improves the work to $O(n \log n)$ by recursively decomposing space into quadrants, and approximating the force due to many far-away bodies (e.g. the gravitational pull of the entire Andromeda galaxy on the Earth) by a single point at their center of mass. Instructors at at Cornell and Princeton have since used this project in their courses.

Through these kinds of examples, I made deterministic parallelism accessible and compelling to my first- and second-year students.

Teaching Philosophy and Practice

To my mind, the most important outcomes of a computer science education are *skills* (knowledge how to do something), rather than *facts* (knowledge that something is true): we should teach students a variety of skills (programming, algorithmic thinking, proof, software system design, user interface design, . . .) that will be useful throughout their careers and further education. I have modeled my approach to teaching skills on Ted Sizer's conception of *coaching*—ask students to practice skills, observe them, suggest improvement, and repeat. I would like to share some examples of how I have integrated coaching into my course.

A prerequisite for coaching is to be clear about what skills students are acquiring. Rather than organizing the course around a long list of concepts and facts (the use of higher-order functions for structuring programs, how parallelism arises from the absence of data dependencies, . . .), I chose to organize the course objectives around three skills: students learn to write parallel functional programs; to analyze sequential and parallel time complexity; and to write and prove mathematical specifications about their code. The facts we want them to learn come up naturally as a substrate for practicing these skills. Importantly, the course interweaves these skills throughout the semester, so that students have time to gradually get better at them. The weekly course activities include two 80-minute lectures, an 80-minute interactive lab, and a homework assignment. To a significant extent, I think of the purpose of lecture as getting students ready for lab, which in turn gets them ready for the homework, which is where they really learn the skills.

In lecture, I introduce new skills in a way that facilitates later coaching on the assignments. For example, I am careful to demonstrate the skills in the way I expect the students to do them—for example, doing a proof at the level of detail I expect students to do on the homework, rather than asking them to be more precise than I am. I also describe and follow explicit methodologies that break a skill down into subtasks, which helps coaching by allowing us pinpoint where a student is stuck. I have also integrated a little coaching even into a large lecture setting: I often make students tell me what to write on the board (and will go down the wrong path, to see why it's wrong, if that's what they suggest), and break up into partners to do exercises (so they coach each other).

While most intro-level classes at CMU have a weekly TA-led lecture (recitation), I chose to use this time instead for interactive coaching. Each week, students come in and work on a miniature homework assignment. The TAs and I circulate around the room, proactively coaching—answering questions when students have them, but also anticipating pitfalls and preemptively making suggestions. I designed labs with opportunities to coach students on all three of the course skills, not just programming: in one crucial lab, students fill in a proof template, so that we can coach them through a particularly tricky new style of proof.

Coaching is integrated into the homeworks in two ways: First, we grade as a group, and read students' code, to provide consistent and useful feedback. Second, the TAs hold tutoring hours in a large classroom, where students can work and ask questions intermittently. One particularly useful opportunity for coaching comes in the capstone of the verification component of the course, when students implement and prove correct a regular expression matcher. This piece of code is very difficult to write, unless you write the program and do its correctness proof at the same time. In the tutoring sessions for this assignment, we coached many students through this proof, and this often led to an "ah ha" moment when they realized what the correct piece of code is, and, more broadly, the importance of verification.

To provide this coaching, I used the staffing I was afforded a little unconventionally: while there is an official distinction between TAs and "course assistants" (CAs), who typically grade and hold office hours only, I asked all the TAs and CAs to do the same duties (though the more experienced people naturally took on more leadership roles). Giving the CAs more responsibility allowed us to have more instructors in lab, and helped foster a TA culture with a lot of camaraderie. It also created a staff pipeline, with new TAs staying on and eventually taking on more leadership roles.

Student Evaluations

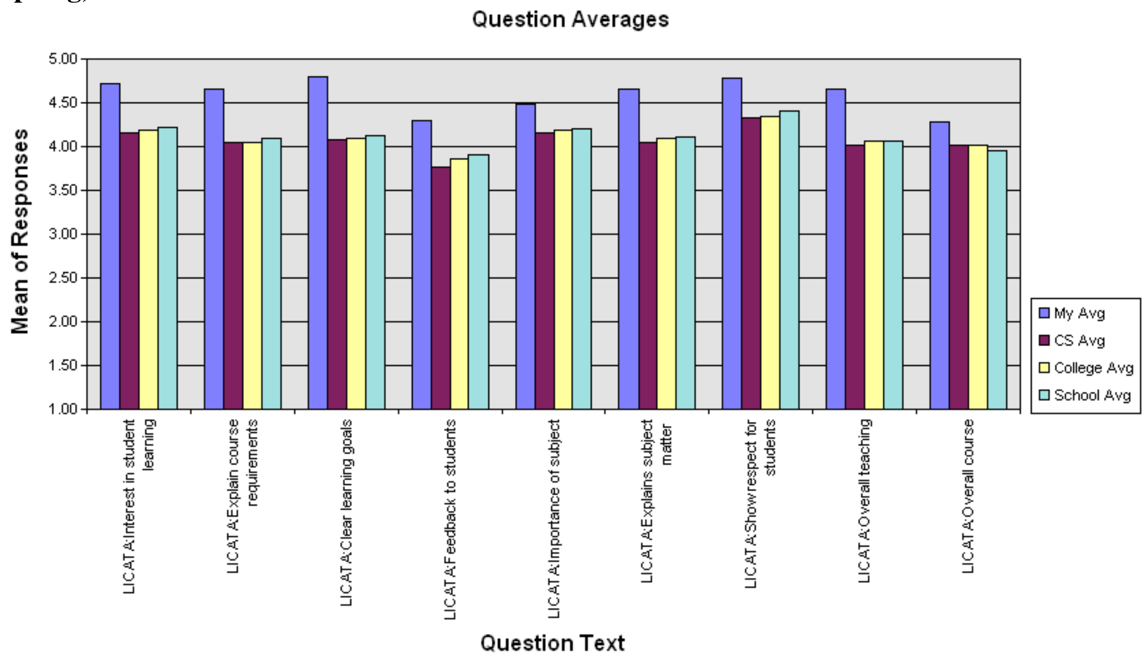
Summary

On CMU's Faculty Course Evaluations, my students repeatedly rated me **above average** to **excellent** on the following criteria, and my scores are almost always **well above the department/college/university averages**.

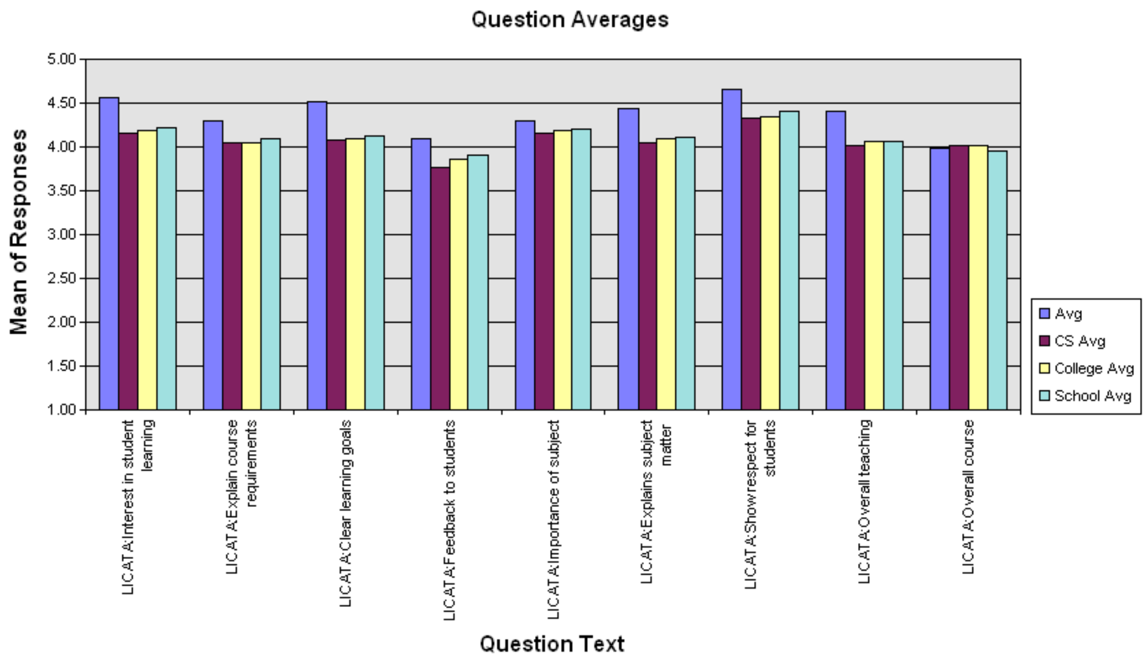
- Displaying an interest in students' learning
- Providing a clear explanation of the course requirements
- Providing a clear explanation of the learning objectives or goals of the course.
- Providing feedback that helped students improve their performance
- Demonstrating the importance and significance of the subject matter
- Explaining the subject matter of the course
- Showing respect for all students
- Overall teaching
- Overall quality of the course

The charts include my average score on these criteria, along with the averages for the CS department, the college (all undergrad classes), and the school (all classes, including graduate) in the specified semester. The blue bar is my score.

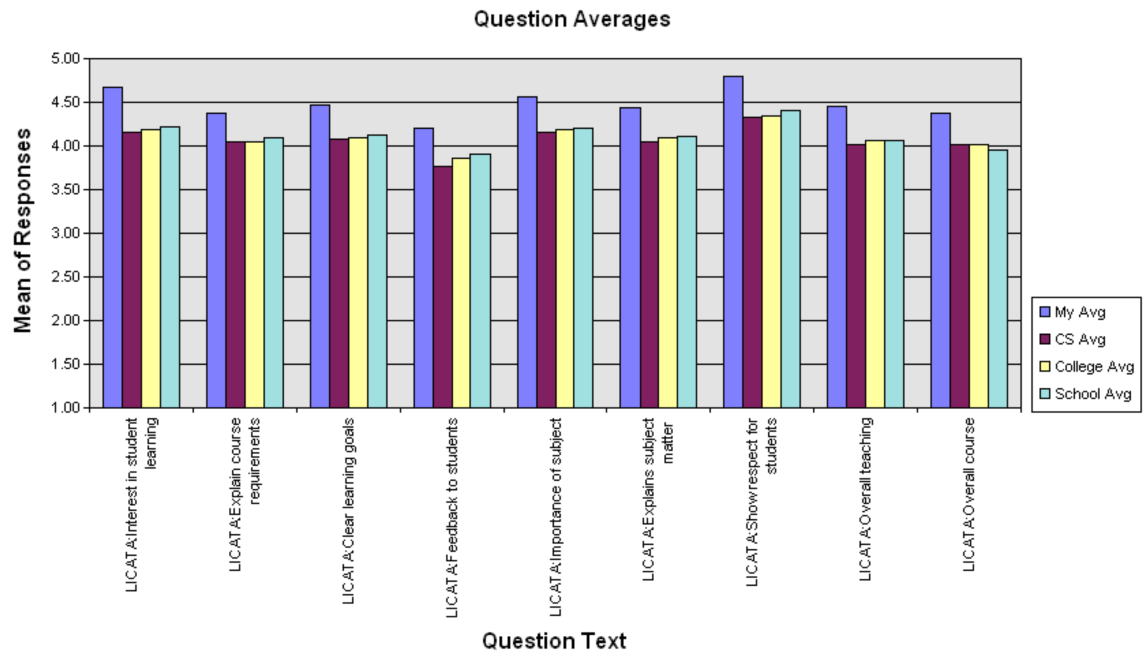
Spring, 2012:



Fall, 2011:



Spring, 2011:



Selected Comments

On lecture:

> *Preface: I'm a Junior here, and to be honest,* ***the 15-150 lecture series was one of the best, if not the best, set of lectures that I've experienced here at CMU.*** *. . . The lectures are absolutely perfect! Please don't change them! =)*

> *. . . great class, great professor, and great TA's. Dan really cares about the subject and you can see that conveyed in his lectures. He clearly knows the material, he knows what he wants to teach and every lecture has a very clear direction. Very organized lectures. Learned a lot.*

> ***It seemed liked every lecture was carefully planed and had a purpose.*** *And I don't think I've ever seen a class with better lecture notes, they're incredibly helpful.*

> *Lectures are actually entertaining (RARE at CMU in CS)*

> *Lectures were simply amazing. Please do not change the lectures. The _only_ thing I might change is to give a little more time on the in-class assignments, because* ***it gets the class excited about the problem at hand / interested in the solution, and it gives them an opportunity to try it out****, but sometimes it'd be nice to have an extra 3-5 minutes. The labs and homeworks are helpful too. The overall structure of 15-150 seems nearly perfect.*

> *This is one class that I do not want to miss a lecture because it is so helpful. That is probably in part Dan's doing, since he is a very good and useful lecturer. Of all the CS classes I've taken, this is definitely the best taught one.*

On assignments:

> ***Labs are also the best out of any class I've taken here and their format should be copied for every lab-based CS class.***

> *I found all of the homeworks and lectures and lab to be very useful. I think the course is structured extremely well, and allows for a great learning experience.*

> *Labs and the homeworks started off easy but built you up to the point where you can tackle more challenging problems.*

> ***I thought this class was really well managed and structured.*** *It was easy to get help, and emails or questions on the bboard are answered promptly. Assignments were well-thought out and clear. Lecture notes were awesome and very well explained. Also, concepts were very clearly explained.*

> *LABS definitely keep the labs. And while the Barnes-Hut was difficult, I liked it. Same with the Connect-4. Programs that work together into something easily testable, and FUN to test, make programming better. For instance, the first time my Connect-4 compiled and ran, it executed one time-step and said "Maxie, you win!" Simple anecdotes like that make larger scale programming a lot more fun, and make the course more fun.*

> *One of my favorite classes. Dan is very approachable, and course staff is very approachable - **unusually and exceptionally useful lab/recitation sessions.** The pace of the class is comfortable, but still fast enough to learn a lot.*

> *Dan's teaching has been very clear. Assignments and labs have been well-integrated with the course material and even though it was difficult at times it was never overwhelming.*

On outcomes:

> ***This class was very challenging, but worthwhile. I feel like I learned quite a lot in a short amount of time.***

> *This was a very challenging course, but definitely my favorite this semester. Dan Licata is an amazing professor, and I hope he teaches future courses so I can can have him again.*

> *Dan was a really great instructor and **helped me become actually interested in the subject matter - I was only taking the course to fulfill a requirement, but I came around to actually enjoying class and the homeworks.***

> *I think this was a fantastic course. I got to learn so much and it was some very exciting material presented in a very organized way.*

> *Professor Licata is a great instructor. He knows the material well and **can easily relate the material to real world applications that makes the understanding of the subject matter very easy**. He also is very willing to help you understand the topics.*

> *Started out slowly for me (since I just took 15-122) but quickly picked up into a comfortably challenging pace. Made me more interested in functional programming, and I will probably take similar classes later on at CMU.*

> *Professor Licata is a great speaker and a dedicated teacher that has always been willing to listen to students' concerns. **He teaches in a well-structured and innovative way, making learning a memorable, applicable hands-on experience in lab, and preparing us for today's and tomorrow's job markets by presenting applications of functional programming in the industry today.** However, the notes he posts are very long.*

On interaction with students:

> *He was always willing to help students outside of class (through Piazza or by attending lab), and **he was very active in explaining difficult concepts and encouraging students who struggled with a particular topic or assignment.***

> *Dan is the best. He's always willing to go above and beyond what's expected of him to help his students, whether that's just re-explaining a topic from lecture or looking through their code at 2AM to help them find a bug they've been looking for for days. The lectures are long, but somehow he manages to keep our attention. It's been such a pleasure to take this class while it was still taught by Dan.*

Professor Licata really treated me well when I was having problems, and I really appreciated that. Even though I did not think his lectures were amazing, ***outside of class he showed an interest in his students learning above what I have experienced previously at CMU.***

Dan did a great job this semester. He was very encouraging. I was very very impressed with ***his understanding and compassion towards the students.***

General:

Dan Licata is the best teacher I've had at CMU, bar none (I'm a senior). If the CS department were serious about educating undergraduates, they would do everything possible to keep and learn from Dan. *He knows how to teach. *This* is how you run a course. More information learned more easily in this course than any other 100 or 200 level CS course.*

Dan Licata is one of the best teachers I've ever had. I knew what I was going to learn, why I was learning it, and felt it was truly something cutting edge and important. *Plus, the teaching was super effective and I feel I know all the stuff. A++ would take class again.*

Dan's probably the best professor I've had since coming to CMU. ***He cares so much about the class and students. I would take a course with him again in a heartbeat.***

This was the best introductory CS course I have taken (out of 15-100, 15-111, and 15-123). Professor Licata was very competent and I could tell that a lot of thought and time was put into the design and teaching of the course. I appreciated that the lectures were taught on the blackboard, instead of with PowerPoint slides.

Okay, so I've rated other teachers as "excellent" before, but I kind of wish I hadn't now because ***Dan really stands out, especially in respect to displaying an interest in student learning and caring about the students.*** *He's super nice and friendly and knows everyone's names and is just an outstanding lecturer in general and one of my favorite professors. He's really helpful if you email him for help too. This is one of my favorite CS classes I've taken, and he's probably a large part of the reason why.*

Dan was actually the best professor I have encountered at this university. ***He was extremely helpful and respectful to all students.*** *I was very happy with his teaching.*

This is my favorite CS course that I have taken at CMU so far. Dan Licata is a wonderful professor and great lecturer, and the class is structured extremely well. This class definitely gets an A+!

Seriously, don't let Dan leave. Or just do what Coachella did and get a high quality hologram of him made. He's hilarious and keeps the lectures engaging. The homeworks were pretty helpful, and I feel like labs prepared me for them well.

Dan Licata is an excellent teacher. It's a shame he's going to Princeton. ***I wish CMU would just hire him now.***