Using a Proof Assistant to Teach Programming Language Foundations
or
Lambda, the Ultimate TA

Benjamin C. Pierce

April 9th, 2008

Abstract

In Fall 2007, I taught an introductory course on logic and the theory of programming languages entirely in Coq. The experience was quite demanding—for the students and especially for me!—but the overall payoff in terms of student engagement and performance on exams far exceeded my hopes. I am now convinced that this is the right way to teach programming language foundations and am working on course materials that will allow the approach to be replicated elsewhere.

Introduction

CIS500 is Penn's introductory programming language theory course at the graduate level. The course has a graduate number and is taken mostly by MSE students, with a few PhDs and undergraduates, but I would expect typical undergraduates at many universities (including Penn) to have if anything an *easier* time with the material than typical MSE students. Thus, it can be taken as a reasonable prototype for an upper-division undergrad PL class.

The course's title is *Software Foundations*, but the actual content is a mixture of basic logic (especially inductive proofs over inductively defined relations), operational semantics, lambda-calculus, and basic material on type systems (simple types, records, exceptions, references, and subtyping, with Featherweight Java as a big case study at the end). The usual textbook is my *Types and Programming Languages*, covering most of the material up through the Featherweight Java chapter.

Naturally, I believe this is all very worthy and important material. But I've long suspected that, for most students, the real value in the course was less the specific PL material than the logical foundations and, more broadly, the rigorous, mathematical approach. Unfortunately, this is also precisely the aspect of the material that I've always felt I've done the *worst* job conveying! I can give most students a cocktail-party level of understanding of operational semantics, preservation and progress theorems, and so on. But when it comes to giving them a really clear idea of when something is a proof and when it is not... well, I give myself a B- at best. The difficulty is that this is essentially a writing problem, and teaching writing is very resource-intensive, requiring thoughtful, detailed feedback on every bit of work handed in by every student. This is simply not possible in a class of 60 or 70 students with one professor and one TA.

The attraction of using an automated proof assistant in this context is obvious: every student can have their own personal TA, 24/7. After hesitating for years, I finally decided last fall that the time had come to take the plunge.

There are several proof assistants now that are very good at what they do, with Coq, Isabelle, and Twelf (in some order :-) arguably leading the pack, at least among PL researchers. Each has its advantages and disadvantages. For this experiment, I chose Coq.[1]

[1]I would choose it again, on the whole, though I'm also attracted by aspects of both Isabelle and Twelf. But, pedagogically

Syllabus

I gave 22 lectures, of 80 minutes each. The first ten minutes of the first lecture were Powerpoint, and the last lecture was a partly formalized sketch (because I ran out of steam); in between, absolutely everything that the students saw[2] was checked Coq scripts.

Proof assistants are fairly complex beasts in their own right. You don't just *add* one to a course: you have to take away some other things to make room. Roughly speaking, I got through a bit more than half of the PL material that I would usually expect to in a semester. On the other hand, I covered logical foundations much more thoroughly than usual, and the students also came out of the course pretty reasonable Coq hackers.

Here, in a bit more detail, is how the semester was organized:

- Functional programming, including higher-order programming and polymorphism, in Coq's built-in functional programming language—basically a strongly normalizing, dependently typed, core of ML. One interesting design choice here was completely ignoring Coq's built-in libraries and constructing everything (numbers, booleans, etc.) from scratch. 2 lectures.
- Simple inductive proofs about functional programs. (It is amazing how many interesting things you can prove using a very minimal set of tactics, if you choose examples carefully!) 3 lectures.
- "Programming with propositions" (i.e., functional programming as logic, via Curry-Howard); induction schemas. Again, I built everything from scratch (including $\vee$, $\wedge$, and $\exists!$), replicating some of the basic functionality of the standard libraries. 3 lectures.
- Operational semantics (of a very simple language of numbers and booleans). Then introducing types for this language and proving progress and preservation. 3 lectures.
- Untyped lambda-calculus. 4 lectures.
- Simple types. 3 lectures.
- Subtyping. 3 lectures.
- Featherweight Java. 1 lecture.

Three class periods were used for review sessions, and three for exams.

Critique

Upsides. Overall, I felt the semester came out very well:

- Students mostly really enjoyed using Coq, found it very motivating (instant feedback, feeling of hacking, etc.), and felt it helped them think more rigorously about what they were doing.
- I gave basically the same exams as in previous years (adjusting the coverage of material to fit the changed syllabus, but asking the same sorts of questions and keeping the difficulty about the same). Although they were being tested on problems that were quite a bit different from the ones they'd worked on in homework (it was impractical to give on-line exams using Coq in this setting, and anyway I felt the potential for people getting completely stuck was too high), the students' performance on these exams was, if anything, better on average than in previous years.

speaking, however, I don't think any of them is quite what I want. For the longer term, I'm very interested in newer projects like Matita that are focusing specifically on problems of pedagogy.

[2]Literally: I lectured from an emacs buffer.

- Before the course, I was quite worried about what was going to happen to the weaker students. I usually feel that there is maybe a third of the class that is really struggling to come to grips with the material (because of weak mathematical training, usually), but that even this group mostly succeeds in getting *something* out of the course. I feared that introducing Coq might (a) increase the size of this group to half the class or more, and (b) make the success criteria so binary that such students would feel they had really gotten nothing out of the experience. In the event, neither thing happened: The group of struggling students seemed about the same as usual, and most of them seemed to get some impression of the material (on the evidence of the their exams).
- The integration of functional programming, logic, and programming language ideas—all revolving around the Curry-Howard Isomorphism—seemed to work very well.
- There was a little less PL-specific material in the course, but I actually didn't mind this — leaving aside from the PhD students planning to work in programming languages, I thought the revamped syllabus was actually more relevant for most students.

Downsides. However, there were also some parts that I was less happy with.

- As I said above, my most important aim in the course was to give students an ability to read and write standard (informal) mathematical proofs. But teaching them to use Coq was teaching them how to do competent *formal* proofs, and these are not at all the same thing. (Just as writing working code and writing design patterns or in other ways communicating the ideas behind a program are not the same thing.) I gave them some sermons about the connection between the two, but I don't think most of them got it.
- It was extremely difficult to shield students from the "deep" aspects of Coq. At the beginning, I was able to identify a small, safe subset that could be explained in an internally consistent way, but as the semester went on this got harder and harder. By the end, there were quite a few subtle and complex topics (Prop vs. Set, reduction strategies, intricacies of the ways induction principles are constructed from inductively definitions of relations, etc., etc.) that we'd dipped our toes into but not really explained.
- Preparing lectures and homeworks was *extremely* demanding for me: I basically did nothing else from early August to mid-December. This was partly because I wasn't a very experienced Coq user myself at the beginning, but mostly because there seem to be between 5 and 20 wrong ways (and a much smaller number of right ways) of formalizing any given PL concept. Another source of stress was the fact that, just as the students knew their homework wasn't finished until Coq said yes, I knew that I couldn't give my lecture until Coq said yes! I didn't actually cancel any lectures because of this, but I was sure I was going to have to, and there were a lot of late nights. Hopefully this will be better next time around.

Outlook

In preparation for offering the same course next Spring, I'm already considering things I'd like to do differently. Here are the main ones:

Reshaping the selection of material. For example, I discovered (*not* to my surprise) that presenting lambda-calculus in a formal way was a bit heavy and fiddly. Indeed (and this *was* a bit of a surprise) the untyped lambda-calculus was much worse than typed systems, particularly because a lot of what I tried to show in this part of the course involved programming *in* the lambda-calculus (church numerals and Y-combinators and all that), and doing this in a formal way involved a lot of coaxing Coq to "animate" large derivations in the operational semantics, which turned out to be surprisingly difficult. With typed lambda-calculi, there is no need to do this, since they are pretty similar to the programming languages people are used to anyway, and one moves directly to the metatheory, which is not so bad.

Still, I think it's worth considering postponing calculi with variable binding until even later in the course. One could spend a couple of weeks, for example, developing denotational semantics for a little while-language. Or, to stay within the operational world, one could formalize a typed assembly language for a "list machine," as Andrew Appel has suggested. Either of these would be much simpler to formalize and easier for the students to come to grips with, and the additional experience would set them up better for a little bit of typed lambda-calculus at the end (probably just skipping over the untyped system).

Another part that I'd skip completely is Featherweight Java: It is simply too complicated, once you get it all formalized—the students get buried in detail.

Better integrating formal and informal proof activities. This is a hard problem, but I think it can be addressed at least partially by getting people to do more exercises of the form "take this formal proof and write a clear informal outline in the form of comments" and "take this outline and generate the formal proof." We did a couple like this late in the course, but the point would be made much better by making it a consistent part of all homework assignments.

More Information

All my lecture materials, homework assignments, and exams can be found here:

`http://www.seas.upenn.edu/~cis500`

If anyone would like to use these materials for teaching, please contact me — I'd love to work with you on improving them, and I can give you access to my "source files," which include numerous comments to myself, suggestions for changes, thoughts about how to present material in lectures, solutions to exercises, etc., etc.