Teaching Discrete Mathematics to Early Undergraduates with Software Foundations

Michael Greenberg
Pomona College
michael.greenberg@pomona.edu

Joseph C. Osborn
Pomona College
joseph.osborn@pomona.edu

1 Introduction

We will present our early experiences teaching first- and second-year computer science students in Coq, adapting *Logical Foundations* (the first volume of *Software Foundations* [2]; henceforth, LF and SF).[1] Our presentation's goals are: to offer evidence that such a course is pedagogically sound; to highlight the pedagogical approach taken to teaching Coq and informal proof simultaneously; to document successes and failures, both pedagogically and in Coq's technical ecosystem; and to seek feedback on pedagogy and course design.

2 CS052 + CS055 ≃ CS054

In Spring 2018, the first author taught an experimental section of a new course, CS054 "Discrete Mathematics and Functional Programming", at Pomona College.[2] The new course replaced two existing courses: a conventional tour of elementary discrete mathematics (CS055) and a second course in computer science touring a variety of programming languages, with a focus on recursion and functional programming (CS052). The primary post-conditions of CS054 are that students ought to be able to:

1. prove theorems by induction;
2. translate between English and propositions;
3. apply basic graph-theoretic terminology; and,
4. program with inductively defined datatypes.

At a minimum: proofs ought to cover the naturals for number theoretic and combinatorial properties; propositions ought to include first-order logic with sets and inductive propositions; inductive datatypes ought to include lists and binary trees.

2.1 New material

Our course covers three new areas compared to the existing pair of courses: combinatorics; a theory of sets and countability; and graph theory. Each posed unique challenges.

Combinatorics and Coq are a fairly natural fit, and others have studied how to use Coq for introductory combinatorics [1]. Haven and Chlipala's approach is too complex (and powerful) for our elementary setting, so we opted to roll our own, simpler definitions of factorial and combinatorial choice. The combinatorial work culminated in two challenging proofs: first, relating the factorial function to permutations of lists; second, proving part of the Binomial Theorem in Coq, adapting the proof to paper, followed by applying the Binomial Theorem on concrete examples.

Set theory in Coq is thornier, with several options for representing sets. We opted for an axiomatic presentation, with students proving set equalities before moving on to countability, which we represented using the usual definitions of injectivity and surjectivity. The most challenging proofs here were (a) finding countability witnesses (explicitly and constructively!) and (b) proving that $|X| < |\mathcal{P}(X)|$ and $|\mathbb{N}| < |\mathbb{N} \to 2|$ in Coq and that $|\mathbb{N}| < |\mathbb{R}|$ on paper.

Finally, graphs are non-inductive and commensurately awkward in Coq. We had low expectations surrounding graphs—students needed to be able to correctly use graph nomenclature (nodes, edges, directed/undirected, paths, etc.) and prove simple properties (e.g, $\sum_{v \in V} \deg(v) = 2|E|$).

2.2 Context: Pomona College

Our students have a variety of backgrounds: some come from elite private high schools and have had courses on proof already; others went to less effective high schools and are mathematical novices.[3] Given our students' wide range of experience, it's critical that students have multiple points of entry to the material. Some students will have thin background not just for mathematics and programming, but for computing in general. We cannot take for granted that students understand ideas like "files and directories in the filesystem" or "the shell".

3 Pedagogy and Content

3.1 Pedagogy in Principle: Formal ⇌ Informal

The pedagogical idea of the course is to interleave formal work in Coq and informal work on the board and on paper. This approach is well supported by educational research [3]: we interleave different presentations of the material; Coq is an oracular tutor for self study and self testing; Coq's `Check` and SF's quizzes encourage explanatory questioning.

[1] The first author will give the talk.
[2] http://www.cs.pomona.edu/~michael/courses/csci054s18/

[3] 17% of Pomona College's students "are in the first generation of their family to attend a four-year college." https://www.pomona.edu/admissions/why-pomona/diversity-pomona.

CoqPL'19, January 19, 2018, Cascais/Lisbon, Portugal
2019.

Our pedagogical approach supports our goal of having more than one way in to the material (Section 2). It can be difficult to predict what a student's way in will be, but zigzagging between the formal and informal helps in two ways. The first way is the natural one: some students take better to more rigid, explicit formulations in Coq, while others prefer the classic mathematical presentation. The second way is simply a side-effect of using Coq: we commonly define things in various ways in Coq to later prove them equivalent.

For example, students define the `evenb` predicate in Basics, the first chapter, along with the `double` function. Later in Logic, the sixth chapter, students prove that n is even iff $\exists k, n =$ `double` k. IndProp, the seventh chapter, relates those definitions to an inductively defined predicate `ev`. Students may or may not see `evenb` as a mathematical definition; they may or may not see $\exists k, n =$ `double` k as the standard "multiple of two" definition of evenness even after proofs relating `double` and `mult`; they may or may not see the inductive `ev` predicate as making a 'counting up from a known even number' argument. Having this plethora of options—and writing and reading proofs of their equivalence both on paper and in Coq—helps students apply the phrasings they understand to the phrasings they do not. While such a multipronged approach could be taken in a conventional class, it is particularly at home in Coq.

3.2 Pedagogy in Practice

Class time alternates formal and informal days. Formal days are mostly lecture from ProofGeneral, using the board to diagram explanations, elaborate examples, run functional programs, and answer questions. Informal days are entirely on the board, with definitions and proofs written in the usual mathematical way, but mostly mirroring our Coq definitions. Early in the course, most new material is introduced formally first, and we often annotate lines in informal proofs with their corresponding tactics. Later in the course, new material is introduced informally and little or no reference is made to Coq concepts. Due to logistical issues, the second run of the course emphasizes the formal parts and has suffered for it.

There's no explicit discussion of proof terms at all in the course. The Curry–Howard correspondence is discussed just enough to support the application of lemmas to arguments. While all students understand the concept of tactics like `apply (plus_comm n m)`, not all necessarily realize that the arrow type in `Set` is one and the same as that in `Prop`.

Assignments in Coq included many informal proofs; additionally, we gave students ungraded practice worksheets, asking questions to structure their understanding (Figure 1).

4 Experience

Overall, we feel the course was effective even in its first run. The first author drew several questions on the final from a final for CS055, the old discrete math course; the old course had 40 students and the new one had 24. Overall student performance on the two finals was comparable: the mean score was 82% in the discrete math course and 81% in the new course.[4] Students performed nearly identically on questions about translating propositions (85% old and 86% new) and constructing truth tables (99% old and 98% new). Student performance worsened significantly on a series of multiple-choice questions about possibly incorrect proofs (72% old and 53% new; some questions were changed). The mean score on an easy inductive proof about sums was 90% in the discrete math course; it was 97% in the new course. Considering it was the first time the course was taught, we interpret these ambivalent results in a positive light: the materials and instruction ought to improve. The multiple-choice questions were particularly anxiety inducing for the students in the new course, and we must aim to increase student confidence.

> Suppose we want to prove that $\forall nm,\ n + m = m + n$.
> 1. What can we do induction on?
> 2. For each possibility above, list (a) the goal you would have to prove in the base case, (b) the induction hypothesis you would get, and (c) the goal you would have to prove in the induction case.
> 3. Which of these inductions would work to prove the theorem?

Figure 1. Sample question from a worksheet on induction

4.1 New material

Compared to the combination of the two old courses, less material is covered. From CS055, the discrete math course, we eliminated discussions of probability and reduced our expectations around combinatorics and number theory. We have retained, however, material on countability. From CS052, the 'CS2' course offering more background in programming, we significantly reduced the programming burden and eliminated material on imperative programming (in Python and a simulated assembly). In exchange, we treat proofs more rigorously, verify programs, and cover inductive propositions.

In Spring 2018, we only lightly adapted SF: we edited prose to suit students who are less mature as programmers and as mathematicians; we added definitions and practice problems, particularly for sums. We haven't significantly amended the course for Fall 2018. The course follows the flow of LF: Basics, Induction, Lists, Poly, Tactics, Logic, and IndProp. We concluded with Sort (from Appel's *Verified Functional Algorithms* volume of SF) and new material on combinatorics, set theory/countability, and graphs.

[4] These numbers are whole-class aggregates in order to support de-identification in compliance with FERPA.

```
Inductive graph : list X -> list (X * X) -> Type :=
| g_empty : graph [] []
| g_vertex :
    forall (V : list X) (E : list (X * X))
           (g : graph V E) (v : X),
      ~ In v V -> graph (v::V) E
| g_arc :
    forall (V : list X) (E : list (X * X))
           (g : graph V E) (src tgt : X),
    In src V -> In tgt V ->
    ~ In (src,tgt) E -> graph V ((src,tgt)::E).
```

Figure 2. An inductive definition for directed graphs, following `graph-basics`

The material on combinatorics was effective but needs to be revised to be simpler. The unit on combinatorics came directly after material on sorting, so we tried to have students relate a function `permutations` : `list` $\alpha \to$ `list` (`list` α)) that computed all permutations of a list and the standard-library inductive proposition `Permutation` : `list` $\alpha \to$ `list` $\alpha \to$ `Prop` that identifies lists that are permutations of each other. Proving that the function generates propositional permutations was not too difficult—though most students couldn't get their heads out of the weeds and see the structure of the proof. Proving that those lists related by the proposition are all found by the function was substantially more challenging—a talented team of three TAs came close, but couldn't finish the proof. We left it as a challenge problem that only one or two students even attempted. The combinatorial proofs involving the Binomial Theorem were too hard. Students had trouble relating conventional sum notation with the higher-order Coq function implementing them. We showed students two ways to work with sums: one by building and then summing lists and one using a higher-order function that takes indices and returns summands. Unifying our treatment and spending time early on relating the formal and informal notations will reduce some of the difficulty.

The work on sets was far too challenging for the time allotted. There are two possible independent courses of action: a computational model of sets (e.g., sorted lists), and more time. Adding a computational interpretation would surely improve student understanding, at the cost of not applying as well to the material on countability. Splitting the sets material into two parts would probably work better: a unit on computational sets where we prove the axioms that we then use to study countability of axiomatic sets.

Finally, we ended up doing graphs entirely informally—students received a paper assignment of questions about drawing graphs and various graph properties. The decision was mostly due to student fatigue after the work on sets. We had prepared material for an inductive notion of graphs, adapted from `graph-basics`[5] (Figure 2). Such an inductive notion greatly simplifies working with graphs. For example, it's a straightforward exerise to prove Euler's handshaking lemma showing that $\sum_{v \in V} \deg(v) = 2|E|$ when both sides of the equation are functions over the graph itself. It's very much *not* straightforward when written over the vertex and edge lists: the proof requires a complex arithmetic lemma relating the sum over the graph and the sum of a map over the vertex list.

4.2 Keeping it simple

Both the tools and the material posed challenges for students.

Students had no problem installing Coq (we used 8.6), but CoqIDE—which we advocated for students without prior Emacs experience—was difficult for them to use. CoqIDE crashes were a common source of woe, but things got particularly difficult with the second assignment, which depends on the first. CoqIDE's conventions for compilation do not work well with the SF single-directory convention. Later on, material on set theory used Unicode characters, which CoqIDE silently mangles while saving to disk. The VS Code plugin for Coq is still too immature, and ProofGeneral's embedding in Emacs presents a serious hurdle. The second author has found that presenting students with a pre-configured Emacs lessens the editor burden and greatly simplifies compilation; conventionalizing Emacs's interface with `cua-mode` would make it still easier.[6]

Gallina is a picayune language, and it's unsurprising that students struggled with some aspects of it. In this case, however, the varying backgrounds and general lack of programming experience was in part an asset: students didn't have expectations like "real programming languages use curly braces" and didn't take things like commutativity of addition and multiplication for granted. Students were quite willing to accept Coq's notions of equality and understood that, e.g., $n + 1 = 1 + n$ is true but not obvious.

SF is a graduate-level course that assumes background in programming and discrete math, spending very little time explaining functional programming or Coq's evaluation model. For example, SF takes it more or less for granted that students are familiar with Boolean operators—but we can make no such assumptions. We added some text, but more is surely necessary. It would be good to rearrange things so that students write more code, defining many of the primitives themselves. SF occasionally takes this approach already, writing some definitions in a `Module` and then later recapitulating Coq's definition or adopting it directly. Such an approach is tricky,: later proofs depend on precise early definitions; adopting Coq's definitions directly can lead to `SearchAbout` including too many results, which less mature students have trouble filtering.

[5] https://github.com/coq-contribs/graph-basics
[6] https://www.emacswiki.org/emacs/CuaMode

4.3 What next?

A novice-friendly IDE would improve the experience of the course; a web-based toolchain would be ideal, with tools like CollaCoq[7], Rhino-Coq[8], and PeaCoq[9] being a nice start, though none are quite mature enough for use with first- and second-year students. It would be a mistake to overrely on tooling, though. Weaker students already struggled with "videogaming" proof search, and better tooling might actually tempt them more.

The course needs some pedagogical revisions. Even after completing a first course in computer science, some students continue to struggle with parsing code; rendering ASTs on the board using Bootstrap's[10] circle notation may help the weakest students early on. The material on inductive propositions is of mixed difficulty, quickly escalating from `ev` up to regular expressions. The treatment of `inversion` will be improved by Prabhakar Ragde's revisions to use `discriminate` and `injection` instead.

Within our curriculum, the course needs much more hands-on programming, since the most interesting programs we write compute list permutations and insertion sort. It's important, though, that the course remain thematically intact: students should write longer programs, but they should also prove interesting properties about them.

Acknowledgments

The first author owes great thanks to his colleagues for allowing this experiment, particularly to the second author who is teaching it in Fall 2018. We also thank Benjamin Pierce and the rest of the Software Foundations authors. CoqPL reviewers had helpful advice on structuring this abstract.

References

[1] Andrew J. Haven. 2013. *Automated Proof Checking in Introductory Discrete Mathematics Classes.* Master's thesis. MIT. Advised by Adam Chlipala.

[2] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Software Foundations.* University of Pennsylvania CIS Department. https://softwarefoundations.cis.upenn.edu/

[3] Henry L. Roediger and Mary A. Pyc. 2012. Inexpensive techniques to improve education: Applying cognitive psychology to enhance educational practice. *Journal of Applied Research in Memory and Cognition* 1, 4 (2012), 242 – 248. https://doi.org/10.1016/j.jarmac.2012.09.002

[7] https://x80.org/collacoq/
[8] https://x80.org/rhino-coq/
[9] http://goto.ucsd.edu/peacoq/
[10] http://www.bootstrapworld.org/