Teaching Discrete Mathematics to Early Undergraduates with Software Foundations

Michael Greenberg and Joseph C. Osborn
Pomona College

CoqPL 2019 @ POPL
Cascais, Portugal
2019-01-19
spoiler alert

- early undergrads can use Coq (CS2)
- they learn real discrete mathematics
- informal and formal proof synergize
- Coq demands careful logistics
my plan for the talk

* background
* pedagogical idea
* what we taught
* evaluation
Consortium
(CMC, HMC, Scripps, Pitzer)

5000 students total
1600 students at Pomona

Majority minority
no-loan policy
12% Pell grants
17% 1st generation
3% undocumented

small classes
more than 30 is big
septem artes liberales

**trivium**: grammar, rhetoric, logic

**quadrivium**: music theory, arithmetic, geometry, astronomy

* students are bright but **may lack background**

* **breadth** over depth

* major is only **11 courses**
existing courses

**CS052**
- 2nd course in the sequence
- functional programming
- tour of CS topics

**CS054**
- functional programming
- recursion
- simple data structures
- automata
- Bayesian reasoning

**CS055**
- pre-req of 4th course (theory)
- discrete math grab bag
- induction
- number theory
- combinatorics
- probability
- graph theory
CS054 goals

* prove theorems by **induction**
* over $\mathbb{N}$ and other structures
* translate between **English** and **propositions**
* **first-order logic** with **sets** and **inductive propositions**
* apply basic **graph**-theoretic terminology
* **program** with **inductively defined datatypes**
* lists and **binary trees**
CS054 non-goals

* become idiomatic Coq users

* understand the Curry-Howard Correspondence
my plan for the talk

* background
* pedagogical idea
* what we taught
* evaluation
pedagogical idea

the **hard part** of learning proof: students don’t know the rules of the game

* Coq enforces the rules...

* ...until students **internalize** them

based on a true story!
my plan for the talk

* background
* pedagogical idea
* what we taught
* evaluation
Basics, Induction, Lists, Poly, Tactics, Logic, IndProp
Logical Foundations
Vol. 1 of Software Foundations (SF)

Sort
Appel’s Verifying Functional Algorithm’s
Vol. 3 of SF

Combo, Sets, Graphs
new!
* lectures mix Coq and chalkboard
* homework in Coq files
  formal and informal proofs
* worksheets not for credit

**two-week topics**

- Induction
- Logic
- Sets

**Basics**

- Lists
- Poly
- Tactics
- IndProp
- Sort
- Combo
- Graphs

**topics over 15 week semester**

**legend**

- all Coq
- mostly Coq
- even mix
- all paper
5. Suppose we want to prove that $\forall nm, n + Sm = S(n + m)$.

(a) What can we do induction on? $n, m$

(b) 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.

Solution: Answer for $n$:
Base case: $0 + Sm = S(0 + m)$
IH: $n' + Sm = S(n' + m)$
Inductive case: $Sn' + Sm = S(Sn' + m)$

Answer for $m$:
Base case: $n + 1 = S(n + 0)$
IH: $n + Sm' = S(n + m')$
Inductive case: $n + S(Sm') = S(n + Sm')$

(c) Which of these inductions would work to prove the theorem? $n$
Exercise: 3 stars (permutation_length)

You may need to define and prove an auxiliary lemma in order to see how everywhere and length interact.

Lemma permutation_length :
∀ A (l l':list A) (a:A),
   In l' (permutations l) → length (everywhere a l') = S (length l).
Proof.
   (* FILL IN HERE *) Admitted.

Exercise: 2 stars (permutations_length)

We can finally prove the desired result: there are factorial n permutations of a list of length n.

Lemma permutations_length :
∀ A (l:list A) n,
   length l = n →
   length (permutations l) = factorial n.
Proof.
   (* FILL IN HERE *) Admitted.
Exercise: 3 stars (lists_of_bools)

How many lists of booleans of length 2 are there? Write them out.

(* FILL IN HERE *)

How many lists of booleans of length 3 are there? No need to write them out.

(* FILL IN HERE *)

Write a theorem characterizing how many many lists of booleans of length \( n \) are there, for any natural \( n \). Your proof should be informal.

(* FILL IN HERE *)
help with tactics

**intros**

Moves things from the goal to the context. It works on quantified variables:

- **FORM:** intros x y z
- **WHEN:** goal looks like forall a b c, H
- **EFFECT:** add x, y, and z to the context (bound to a, b, and c, respectively); goal becomes H
- **INFORMAL:** "Let x, y, and z be given."

**destruct**

Performs case analysis. Its precise use depends on the inductive type being analyzed. Be certain to use - / + / * to nest your case analyses. Always write an as pattern.

- **FORM:** destruct n as [ | n']
- **WHEN:** n : nat is in the context
- **EFFECT:** proofs splits into two cases, where n=0 and n=S n' for some n'
- **INFORMAL:** "By cases on n. - If n=0 then... - If n=S n', then..." If you're at the beginning of a proof, don't forget to "let n be given". It's often good to say what your goal is in each case.
## CS054 — How to prove it

Text in black is the “script”—it stays the same every time; text in monospace is the corresponding Coq code. Text in red is the rest of proof—have to figure that part out!

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Pronunciation</th>
<th>How to prove it</th>
<th>How to use it</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀x, P(x)</td>
<td>for all x, P(x)</td>
<td>Let x be given. Now prove P(x) for this arbitrary x we know nothing about. intros x</td>
<td>We have y and know ∀x, P(x); therefore, P(y). apply .../apply ... in ...</td>
</tr>
<tr>
<td>∃x, P(x)</td>
<td>there exists an x such that P(x)</td>
<td>Let x = choose some object, y. Now prove P(y) for your choice of y. exists ...</td>
<td>We have ∃x, P(x), so let y be given such that P(y). destruct ... as [x H]</td>
</tr>
<tr>
<td>p ⇒ q</td>
<td>p implies q; if p, then q</td>
<td>Suppose p. Now prove q, having assumed p. You don’t have to prove p. intros H</td>
<td>Use #1: We have p ⇒ q; since proof of p, we have q. apply ... in ... Use #2: We must show q, but we have p ⇒ q, so it suffices to show p. Now go prove p! apply ...</td>
</tr>
<tr>
<td>p ∧ q</td>
<td>p and q</td>
<td>Prove p. Prove q. split</td>
<td>We have p ∧ q, i.e., we have both p and q. destruct ... as [Hp Hq]</td>
</tr>
</tbody>
</table>
| p ∨ q        | p or q        | **Proof #1:** To see p ∨ q, we show p. Prove p. You don’t have to prove q. left | We have p ∨ q. We go by cases. 
(p) If p holds, then prove whatever your goal was, given p. Ignore q. 
(q) If q holds, then prove whatever your goal was, given q. Ignore p. destruct ... as [ Hp | Hq] |
| ¬p           | not p         | To show ¬p, suppose for a contradiction that p holds. Now find a contradiction, like 0 = 1 or q ∧ ¬q or 5 < 1. intros contra; destruct/inversion | We have ¬p; but proof of p—which is a contradiction. Now you’re done with whatever case you’re in! exfalso; destruct/inversion |

**Derived forms**

| p ↔ q        | p iff q; p if and only if q | We prove each direction separately: 
(⇒) Suppose p; proof of q. 
(⇐) Suppose q; proof of p. | Use #1: We have p ↔ q; since proof of p, we have q. Use #2: We have p ↔ q; since proof of q, we have p. |
| ∀x, P(x) ⇒ Q(x) | for all x such that P(x) holds, Q(x) holds | Let an x be given such that P(x). Prove Q(x), given that P(x) holds. | Choose some y. Since we have P(y), we can conclude Q(y). |
| ∀x ∈ S, P(x) | for all x in S, P(x) holds | Let an x ∈ S be given. Prove P(x), given that x is in the set S. | Choose some y ∈ S. We have P(y). |
my plan for the talk

* background
* pedagogical idea
* what we taught
* evaluation
what worked

- nearly same exam in CS054, CS055
- nearly same mean score

- too hard and too much material
- students still had fun

<table>
<thead>
<tr>
<th></th>
<th>CS055</th>
<th>CS054</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP</td>
<td>20%</td>
<td>25%</td>
</tr>
<tr>
<td>MID</td>
<td>60%</td>
<td>55%</td>
</tr>
<tr>
<td>COQ</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>BOT</td>
<td>20%</td>
<td>5%</td>
</tr>
</tbody>
</table>
what didn’t work

- CoqIDE was a nightmare
  - **crashy**, non-native UI
  - silently mangling **Unicode** characters
  - **bad defaults** for .vo files... needed CLI
- no **documentation** at an appropriate level
- **grading informal proofs** was awkward
- **new material** was rough
- no time to try **graphs formally**
challenges

* set theory desiderata:

* executable operations on any type in Set
* potentially infinite
* allow for a treatment of countability
* less arithmetic drudgery
* more interesting total programs
sets

* **axiomatized** naïve typed set theory
  
  set : Type -> Type  
  Universe : forall {X}, set X

* students did a bunch of **equational proofs**
  
  e.g., **De Morgan's laws**

* **countability** just on Coq's types
  
  formal proofs included:
  
  |nat| = |list unit| = |option nat|  
  |nat| ≤ |nat -> nat|  
  |x : set T| ≤ |power_set(x) : set (set T)|

  informal proofs included: |N| ≤ |Q|, etc.
wrote up inductive graphs following Jean Duprat’s GraphBasics

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).

proving Euler’s Handshaking Lemma is easy: \[ \sum \deg(v) = 2 \cdot |E| \]

...much harder with standard maps and sums on V and E!
what's next

* use *emacs*, spend a day on it and the CLI
* formal/informal as *separate submissions*
* ...but keep the questions *related!*
* uniform treatment of *sums*
* more programming
je suis un peu difficile...