Learning Support for Writing Proofs in Coq

Jeremy Yew
MCS Capstone presentation, 13/11/19
Advisor: Professor Olivier Danvy

Learning Support for Writing Proofs in Coq

Coq: A system for writing and verifying proofs

- YSC3236 Functional Programming and Proving (FPP).
  - Target audience: FPP students, who are new to Coq.
- Learning philosophy: training in a skilled discipline.
  - Prescriptive.
  - Build muscle memory.
- Rigorous, progressive exercises.
  - Practice specific proof techniques and programming habits.
- Students independently write hundreds of proofs.
- They also learn to state theorems.

Learning Support for Writing Proofs in Coq

Writing proofs: What could go wrong?

- Multiple representations of programs and proofs.
  - Arising from flexibility.
  - Counterproductive for beginning learners.
- Abuse of tactics.
- Misuse of tactics.

What could go wrong: Abuse of tactics

```
Lemma SSSn_is_3_plus_n :
  forall n : nat,
  S (S (S n)) = 3 + n.
Proof.
  intro n.
  rewrite <- (Nat.add_1_l n).
  rewrite <- (plus_Sn_m 1 n).
  rewrite <- (plus_Sn_m 2 n).
  reflexivity.
Qed.
```

```
Lemma SSSn_is_3_plus_n :
  forall n : nat,
  S (S (S n)) = 3 + n.
Proof.
  trivial.
Qed.
```

If you can't explain what you're doing, you don't understand it.

What could go wrong: Misuse of tactics

```
Check Nat.add_assoc.
# Nat.add_assoc : forall n m p : nat, n + (m + p) = n + m + p.
```

```
Proposition add_assoc_nested :
  forall a b c d e: nat,
    a + b + c + d + e =
    a + (b + (c + (d + e))).
Proof.
  intros a b c d e.
  rewrite -> (Nat.add_assoc a b (c + (d + e)) ).
  rewrite -> (Nat.add_assoc (a + b) c (d + e) ).
  rewrite -> (Nat.add_assoc (a + b + c) d e ).
  reflexivity.
Qed.
```

```
Proposition add_assoc_nested :
  forall a b c d e: nat,
    a + b + c + d + e =
    a + (b + (c + (d + e))).
Proof.
  intros a b c d e.
  rewrite -> (Nat.add_assoc a b )
  rewrite -> (Nat.add_assoc (a + b) ).
  rewrite -> Nat.add_assoc.
  reflexivity.
Qed.
```

Learning Support for Writing Proofs in Coq

Learning Support: A syntax parser

- Program to enforce explicit tactic application within a given subset of Coq.
  - Input: student's Coq files, and a grammar specification.
  - Output: warnings about instances of rule violation.
- Integrated with Emacs editor for interactive use.
- 'Safety rails' for first half of the course.
- Earlier intervention.
- Substantive rather than superficial feedback.

Learning Support: Implementation trade-off

- Custom parser vs Parser generator?
  - Don’t reinvent the wheel.
  - Minimal code.
  - Outputs Emacs Lisp code.
  - More extensible.
- Cons:
  - Need to learn grammar notation.
  - Poor documentation.

Learning Support: Progress

- Work so far.
  - Defining a subset of Coq grammar.
  - Different types of parsers for different types of languages.
  - How to write a parser.
  - How to write Emacs Lisp programs.
  - How to write an Emacs extension.
- Challenges ahead.
  - Implement rules to address two issues.
  - Explore more rules, interactive features.
- Iteration and Success:
  - Feedback from Professor and students on usability.

Related work/resources

- Glickstein, Bob. *Writing GNU Emacs Extensions*. O'reilly Media, Inc., 2010.
- Coq reference manual: The Gallina specification language https://coq.inria.fr/distrib/current/refman/language/gallina-specification-language.html
- Bovine grammar rules https://www.gnu.org/software/emacs/manual/html_node/bovine/Bovine-Grammar-Rules.html
- Bovine grammar example https://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html

Takeaways

Coq

- Training in a skilled discipline.
- Rigorous, progressive exercises.
- Muscle memory, good habits.

Writing Proofs

- Abuse of tactics (‘magical’ tactics).
- Misuse of tactics (rewrite rule arguments).

Learning Support

- Explicit tactic application within a subset of Coq.
- Integrated with Emacs editor.
- ‘Safety rails’.
- Earlier intervention.
- Substantive rather than superficial feedback.