Learning Support for Mechanizing Proofs in Coq

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

Learning Support for Mechanizing Proofs in Coq

Coq: A language for mechanical proofs

- Natural language proof
  - Informal, unstructured. Convention and formal notation mitigate ambiguity, but unenforced and inconsistent.
  - Fallible human verification.
- Mechanical proof
  - Formal, structured.
  - Automated machine verification.
- Coq interactive theorem prover
- Context: YSC3236 Functional Programming and Proving (FPP)
  - Target audience: FPP students, who are new to Coq.
  - Learning goal: to build muscle memory via rigorous, prescriptive training in order to develop good programming and proving habits.
  - Students independently write hundreds of proofs.

Learning Support for Mechanizing Proofs in Coq

Mechanizing proofs: What could go wrong?

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

What could go wrong: Breaking structural conventions

```
Lemma disjunction_is_commutative :
  forall A B : Prop,
    A \/ B -> B \/ A.
Proof.
  intros A B.
  intros [H_A | H_B].
  - right.
    exact H_A.
  - left.
    exact H_B.
Qed.
```

```
Lemma disjunction_is_commutative :
  forall A B : Prop,
    A \/ B -> B \/ A.
  intros A B.
  intros [H_A | H_B].
Proof.
  - right.
    exact H_A.
  - left.
    exact H_B.
Qed.
```

```
Proof.
Lemma disjunction_is_commutative :
  forall A B : Prop,
    A \/ B -> B \/ A.
  intros A B.
  intros [H_A | H_B].
  - right.
    exact H_A.
  - left.
    exact H_B.
Qed.
```

What could go wrong: Abuse of tactics

```
Lemma disjunction_is_commutative :
  forall A B : Prop,
    A \/ B -> B \/ A.
  intros A B.
  intros [H_A | H_B].
Proof.
  - right.
    exact H_A.
  - left.
    exact H_B.
Qed.
```

```
Lemma disjunction_is_commutative :
  forall A B : Prop,
    A \/ B -> B \/ A.
  intros A B.
  intros [H_A | H_B].
Proof.
  intuition.
Qed.
```

What could go wrong: Misuse of tactics

```
Lemma addition_is_commutative :
  forall n m: nat,
    n + m = m + n.
Proof.
  induction m as [ | m IHm ].
  Search (_ = _ + 0).
  rewrite -> (plus_O_n n).
  rewrite <- (plus_n_O n).
  reflexivity.
  ...
```

```
Search (_ = _ + 0).
# plus_n_O: forall n : nat, n = n + 0
```

```
Lemma addition_is_commutative :
  forall n m: nat,
    n + m = m + n.
Proof.
  induction m as [ | m IHm ].
  Search (_ = _ + 0).
  rewrite -> plus_O_n.
  rewrite <- plus_n_O.
  reflexivity.
  ...
```

Learning Support for Mechanizing Proofs in Coq

Learning Support: A syntax parser

- Motivating assumptions
  - Bad style reflects disorganized thoughts, lack of understanding, and will impact future work.
  - Good style improves clarity of thinking and program readability.
- Program to enforce structural conventions and explicit tactic application within a subset of Coq.
  - ‘Safety rails’ for first half of the course.
  - Input: student’s Coq files, set of syntax rules.
  - Output: warnings about instances of rule violation.
  - Integrated with Emacs editor for interactive usage.

Learning Support: Implementation trade-offs

- External language vs Emacs Lisp
  - Using an external language (Python, Java, etc) might be quicker to obtain first working version.
  - Emacs Lisp runs in Emacs editor, and avoids interoperability issues, and is part of the ecosystem.
- Custom parser vs Parser generator
  - Don’t reinvent the wheel.
  - Parser generator would require minimal code, and outputs Emacs Lisp code.
  - More extensible - declarative grammar notation as opposed to hard-coded program branches.

Learning Support: Implementation trade-offs

- Sample Bison grammar notation for an infix calculator:

```
exp:
 NUM
| exp '+' exp          { $$ = $1 + $3;      }
| exp '-' exp          { $$ = $1 - $3;      }
| exp '*' exp          { $$ = $1 * $3;      }
| exp '/' exp          { $$ = $1 / $3;      }
| '-' exp  %prec NEG   { $$ = -$2;          }
| exp '^' exp          { $$ = pow ($1, $3); }
| '(' exp ')'          { $$ = $2;           }
;
```

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 three rules.
  - Explore more rules, interactive features.
- Success
  - Working version would be successful based on motivating assumption.
  - 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

Learning Support for Mechanizing Proofs in Coq

Coq: A domain-specific language for constructing machine-verifiable proofs, with which FPP students independently write hundreds of proofs.

Mechanizing Proofs: Reinforcing good habits to build muscle memory (breaking structural conventions, abuse and misuse of tactics).

Learning Support: Building an Emacs-integrated language parser that enforces additional syntax rules, either programmed by hand or with a parser generator.