Interactive Theorem Proving with Coq

Adam Chlipala

Motivation

- We focus mainly on automated deduction in this class.
- There are many interesting theories that we don't yet know how to decide automatically. For instance:
  - Formalizing large parts of traditional math
  - Or proving the soundness of particular proof-carrying code systems

Outline

- Come up with a suitably general encoding for propositions and proofs
- See how systems like Coq can make it easier to generate formal proofs
- Revisit a past lecture by using Coq to prove the correctness of JML-annotated Java programs
- Go in the opposite direction by translating Coq proofs into executable ML programs

Proof checking via type checking

- Recall the discussion of proof representation in an earlier lecture.
- We can express logical propositions with an ML-style datatype.
- If we add dependent types, we can even express deduction rules as terms.
- A supposed proof proves some proposition only if it *type-checks to have that proposition's type*.

Review: Conjunction

$$and : prop \to prop \to prop$$

$$\frac{A \quad B}{A \wedge B} \wedge I$$

$$andi : \Pi A : prop.\ \Pi B : prop.\ A \to B \to (and\ A\ B)$$

$$\frac{A \wedge B}{A} \wedge E1$$

$$ande1 : \Pi A : prop.\ \Pi B : prop.\ (and\ A\ B) \to A$$

Enter the ML type checker!

- The other propositional connectives can be described with similar-looking terms.
- While ML doesn't support dependent types in general, the types for propositional proof constructors all fit into a format that it *does* support.
- Instead of defining a new type of propositions, we can use the language of ML types itself as our proposition type!
- ML polymorphism allows quantification over types.

Demo: Proof checker

- This means that every ML compiler already contains the essential machinery for checking a complete proof system for propositional logic!

See demo....

But is all that necessary?

- ML contains many more features than would be required if we just wanted a proof checker.
- Also, it's not clear whether it would support all new logical formalisms we might come up with.
- Coq uses the *Calculus of Inductive Constructions* (CIC), a system powerful enough to allow the definition of the logical connectives using a simple extension of lambda calculus.

CIC

- Start with the simply typed lambda calculus.
- Add dependently-typed polymorphism.
- Add a way to define recursive data types and primitive recursive functions over them.
- These features are all that 99% of Coq developments use.

Defining connectives

```
Inductive and
    : Prop -> Prop -> Prop :=
  | andi : forall (A B : Prop),
      A -> B -> and A B.

Inductive or
    : Prop -> Prop -> Prop :=
  | ori1 : forall (A B : Prop),
      A -> or A B
  | ori2 : forall (A B : Prop),
      B -> or A B.
```

Defining equality

```
Inductive eq
    : forall (T:Type), T -> T -> Prop
  | eqi : forall (T : Type) (X : T),
      eq X X.
```

Interactive proving

- Coq works mostly using backwards reasoning.
- You begin a proof by specifying a goal to be proved.
- You specify a series of tactics that in general produce multiple sub-goals with different sets of hypotheses.

See demo....

Proving program correctness

- In a past lecture, we saw how to use ESC/Java to find many bugs in Java programs.
- We also saw many ways to trick ESC/Java into accepting buggy programs. ☺
- We've seen how to produce verification conditions for programs annotated with specifications.
- However, today's automated tools are generally not clever enough to prove these conditions.

Manual correctness proofs

- *Krakatoa* is a verification condition generator for Java programs annotated with JML.
- It can generate a series of Coq lemma statements that together imply that that a Java program meets its spec.
- A human has to go through and prove the tricky parts of these lemmas.

Benefits

- If you can prove all of the lemmas, then you can be sure that the program meets its specification.
- There is no chance that a bug-finding tool's heuristics just weren't smart enough to find a bug.

See demo for insertion sort....

Compiling proofs into programs

- Most Coq proofs use *constructive logic*.
- It is well-known that such proofs have computational interpretations.
- The early example of propositional-logic-in-ML should give some of the intuition behind this.

Programming by proving

- This means that it is possible to develop a program by proving that its specification is satisfiable!

See demo....