Interactive Computer Theorem Proving

Lecture 11: Proof by Reflection

CS294-9
November 2, 2006
Adam Chlipala
UC Berkeley

Proofs Revisited

1 + 1 = 2

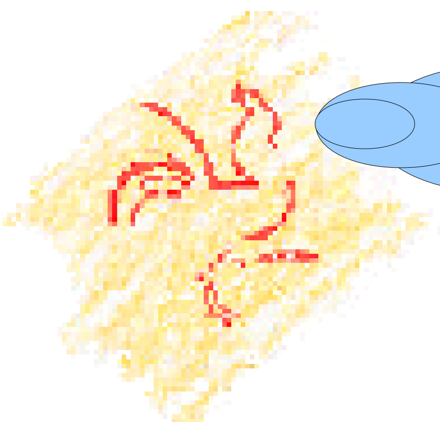
...in standard first-order logic

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{=I}
\UnaryInfC{$\mathrm{S\ O + S\ O = S\ O + S\ O}$}
\AxiomC{}
\RightLabel{Def.}
\UnaryInfC{$\forall\, n, m,\ \mathrm{S}\ n + m = n + \mathrm{S}\ m$}
\RightLabel{$\forall$E}
\UnaryInfC{$\forall\, m,\ \mathrm{S\ O} + m = \mathrm{O} + \mathrm{S}\ m$}
\RightLabel{$\forall$E}
\UnaryInfC{$\mathrm{S\ O + S\ O = O + S\ (S\ O)}$}
\RightLabel{=E}
\BinaryInfC{$\mathrm{S\ O + S\ O = O + S\ (S\ O)}$}
\AxiomC{}
\RightLabel{Def.}
\UnaryInfC{$\forall\, n,\ \mathrm{O} + n = n$}
\RightLabel{$\forall$E}
\UnaryInfC{$\mathrm{O + S\ (S\ O) = S\ (S\ O)}$}
\RightLabel{=E}
\BinaryInfC{$\mathrm{S\ O + S\ O = S\ (S\ O)}$}
\end{prooftree}
$$

Proofs Revisited

1 + 1 = 2

...in Coq



Evaluate the proof fully before checking
correct rule usage....

S O + S O = S (S O)
O + S (S O) = S (S O)
S (S O) = S (S O)

OK, now it's clear that reflexivity
justifies this!

$$\frac{}{\text{S O + S O = S (S O)}}\;=\text{I}$$

A Simple Motivating Example

$$\frac{}{\text{isEven O}}\ \text{Even\_O} \qquad \frac{\text{isEven } n}{\text{isEven S (S } n\text{)}}\ \text{Even\_SS}$$

To prove $\text{isEven } (2\ n)$:

$$\underbrace{\text{Even\_SS (Even\_SS (Even\_SS (Even\_SS (Even\_SS}}_{n \text{ Even\_SS's}} \text{ (...Even\_O...)))))}$$

In the real representation, the variable n of Even_SS appears explicitly in each rule invocation, so we have a quadratic-size proof scheme!

Challenge: What proof scheme allows us to build isEven proofs for constants such that there exists some constant K where the proof size for $2n$ is only K more than the size of the representation of $2n$?

Reflection Recipe (First Cut)

To prove goals that use a predicate P:

1. Implement in Coq a (partial) decision procedure for P that approximates the truth of P with an algorithm returning booleans.
2. Prove that P holds when the procedure returns true.
3. Prove individual instances by applying that soundness theorem with reflexivity proofs.

An isEven Decision Procedure

```
Fixpoint check_even (n : nat) : bool :=
  match n with
    | 0 => true
    | 1 => false
    | S (S n') => check_even n'
  end.
```

Key property: For any constant n, check_even n evaluates to true or false, using only Coq's built-in reduction rules.

Soundness Theorem

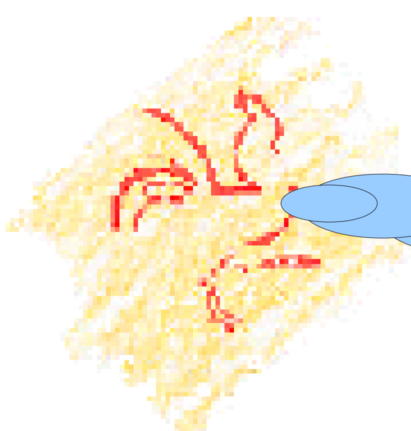
Theorem check_even_sound : **forall** n,

check_even n = true

-> isEven n.

Generic proof of isEven $2n$:

check_even_sound $2n$ (refl_equal true)



check_even_sound $2n$:

check_even $2n$ = true -> isEven $2n$

Reduce to:

true = true -> isEven $2n$

Reflective “Tauto”

$$(P1 \lor Q1) \land (P2 \lor Q2) \land \ldots \land (Pn \lor Qn)$$
$$\to (Q1 \lor P1) \land (Q2 \lor P2) \land \ldots \land (Qn \lor Pn)$$

The tauto tactic (which is also used by intuition) solves this goal by expanding out all 2^n cases arising from the disjunctions in the assumption, leading to exponentially-sized proofs.

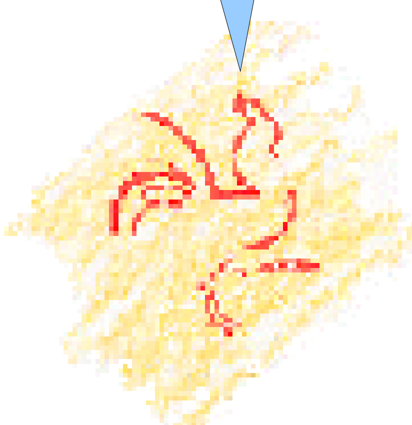
Challenge: Develop a reflective proof scheme that lets us prove formulas in a useful class of tautologies with proof size **quadratic** in the formula's length.

First Attempt

To keep it simple, let's start by considering formulas built from True, False, and, and or.

> **Prop** isn't an inductive type!

```
Fixpoint eval_formula (P : Prop) : bool :=
  match P with
   | True => true
   | False => false
   | P1 /\ P2 => eval_formula P1 && eval_formula P2
   | P1 \/ P2 => eval_formula P1 || eval_formula P2
   | _ => false
  end.
```



Revised Reflection Recipe

To prove goals that use a predicate P:

1. Create a **syntactic** representation S of P's domain D.
2. Define a **compilation** of S into D.
3. Implement a decision procedure over S and prove that, when it returns true for s in S, P holds of the **compilation** of s.
4. Use the soundness theorem reflectively.

A Syntactic Representation

```
Inductive formula : Set :=
  | Truth : formula
  | Falsehood : formula
  | And : formula -> formula -> formula
  | Or : formula -> formula -> formula.
```

```
Fixpoint interp_formula (f : formula) : Prop :=
  match f with
    | Truth => True
    | Falsehood => False
    | And f1 f2 => interp_formula f1 /\ interp_formula f2
    | Or f1 f2 => interp_formula f1 \/ interp_formula f2
  end.
```

A Prover

```
Fixpoint eval_formula (P : formula) : bool :=
  match P with
    | Truth => true
    | Falsehood => false
    | And P1 P2 => eval_formula P1 && eval_formula P2
    | Or P1 P2 => eval_formula P1 || eval_formula P2
  end.
```

Its Soundness Theorem

Theorem eval_formula_sound : **forall** (f : formula),

eval_formula f = true

-> compile_formula f.

Generic scheme for proving formula P:

Compute the **representation** f of P.
(This step can't be done in Coq's logic!)

Use the proof term:

eval_formula_sound f (refl_equal true)

An Example

Want to prove:

True $\vee$ False

Representation:

Or Truth Falsehood

Proof term:

eval_formula_sound (Or Truth Falsehood) (refl_equal true)

eval_formula (Or Truth Falsehood) = true

-> compile_formula (Or Truth Falsehood)

compile_formula (Or Truth Falsehood)

Reduce

True $\vee$ False

Reduce

true = true

-> compile_formula (Or Truth Falsehood)

Second Version

Now let's expand our class of formulas to include, in addition to True, False, and, and or:

- Implication
- Arbitrary propositions as “propositional variables”

```
                    Type
Inductive formula : ~~Set~~ :=
  | Truth : formula
  | Falsehood : formula
  | And : formula -> formula -> formula
  | Or : formula -> formula -> formula
  | Imp : formula -> formula -> formula
  | Atomic : Prop -> formula.
```

Not Quite There….

Now we can prove some theorems:

```
Definition easy_prover (f : formula) : bool :=
  match f with
   | Or True _ => true
   | _ => false
  end.
```

But we get stuck for other “easy” ones:

Example: How can we write a prover that can show:

P -> P

for any P?

Problem: We can't compare **Props** for equality

Third Version

```
Inductive formula : Set :=
  | Truth : formula
  | Falsehood : formula
  | And : formula -> formula -> formula
  | Or : formula -> formula -> formula
  | Imp : formula -> formula -> formula
  | Atomic : var -> formula.
```

On the side, maintain a mapping:

$\mathrm{atomics} : \mathrm{var} \texttt{->} \mathbf{Prop}$

We will depend on decidable equality for var:

$\mathrm{var\_eq\_dec} : \mathbf{forall}\ (x\ y : \mathrm{var}),\ \{x = y\} + \{x <> y\}$

It Works!

```
Fixpoint interp_formula (f : formula) : Prop :=
    match f with
      | Truth => True
      | And f1 f2 => interp_formula f1 /\ interp_formula f2
      | Atomic v => atomics v
     ...
    end.
```

```
Fixpoint prover (f : formula) : bool :=
    match f with
      | Imp (Atomic v1) (Atomic v2) =>
        if var_eq_dec v1 v2
            then true
            else false
      ...
    end.
```