Interactive Computer Theorem Proving

Lecture 7: Programming with Proofs

CS294-9
October 5, 2006
Adam Chlipala
UC Berkeley
Classical Program Verification

Verification Condition Generator

A \land B \rightarrow C

Verification Condition

A' \land B' \rightarrow C'

Interactive Provers
(Coq, Isabelle, PVS, etc.)

Proof
Problem #1: Error diagnosis during evolution

Definition foo := ...!
Definition bar := ....

Lemma L : forall x, P (foo x) (bar x).
  tactic1.
  tactic2.
Qed.

Theorem T : forall x, Q (foo x) (bar x).
  tactic1.
  tactic2.
  ...
  tactic3253.
Qed.
Problem #2: Effective Modular Development

**Definition** foo := ....

**Definition** bar := ....

**Import** BeppoLib.

**Definition** baz := ... foo ... bar ....

**Definition** baz' := ... foo ... bar ... beppo ....

**Theorem** foo_correct.

....

Qed.

**Theorem** bar_correct.

....

Qed.

**Theorem** baz_correct.

....

apply foo_correct.

....

apply bar_correct.

....
Internal Verification

h

Precondition and postcondition

Proof of precondition
Proof of postcondition

x
y
f

Precondition and postcondition

u
g

Proof of postcondition
Proof of precondition

Precondition and postcondition

Proof of postcondition

...
Problem #1: Verbosity

Proof added as a subgoal for tactic-based proving....

Proof of postcondition
Proof of precondition
Problem #2: Runtime Inefficiency

- Manipulating proof objects at runtime will slow programs down a lot.
- Some simple algorithms have much larger and more complicated proofs, so proof processing would dominate performance.
- Your computer probably doesn't have enough RAM to store explicit proof objects arising from large inputs to interesting programs.
The Big Idea:
Proofs are a technical device for establishing program soundness statically. There's no reason to include them in the final programs to execute!

We want some kind of proof-erasing compiler.

Challenges:
• What do we cut and what do we keep?
• How can we ensure compilation soundness?
Prop vs. Set

• We've come to the sole reason for distinguishing between types in Prop and Set: Prop values are erased during compilation, while Set values are kept.

• If we didn't want extraction, we could collapse them into a single domain.

With the default flags, Set and Prop are also differentiated by predicativity.
A Tale of Two Sorts

**Inductive** unit : Set :=
   tt : unit.

**Inductive** Empty_set : Set := .

**Inductive** prod (A B : Set) : Set :=
   pair : A -> B -> prod A B.

**Inductive** sum (A B : Set) : Set :=
   inl : A -> sum A B
   | inr : B -> sum A B.

**Inductive** True : Prop :=
   I : True.

**Inductive** False : Prop := .

**Inductive** and (A B : Prop) : Prop :=
   conj : A -> B -> and A B.

**Inductive** or (A B : Prop) : Prop :=
   or_introl : A -> or A B
   | or_intror : B -> or A B.
The Coq Type Hierarchy

\[
\begin{array}{c}
\cdots \\
\downarrow \\
\text{Type}_1 \\
\downarrow \\
\text{Type}_0 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Set} \\
\downarrow \\
nat \quad \text{bool} \quad \ldots \\
\end{array}
\quad
\begin{array}{c}
\text{Prop} \\
\downarrow \\
\text{True} \quad \text{False} \quad \ldots \\
\end{array}
\]
What Coq Extraction Does

- Translate Coq code to OCaml, Haskell, or Scheme (*straightforward part*)

- Systematically replace all Props with unit, so that their values carry no usable information (*tricky part*)

Example:

**Definition** \( f : \forall (n : \text{nat}), n > 0 \rightarrow \text{nat} := \ldots \)

Erase Prop

**Definition** \( f : \forall (n : \text{nat}), \text{unit} \rightarrow \text{nat} := \ldots \)

Simplify types involving unit

**Definition** \( f : \forall (n : \text{nat}), \text{nat} := \ldots \)

Drop dependent typing and translate to OCaml syntax

\text{let} \( f : \text{nat} \rightarrow \text{nat} := \ldots \)
Related Problem: Secure Information Flow

Ensure that no public output depends on a secret input.
Related Problem: Taint Analysis

Ensure that no `printf` format string depends on a user input.
The Essence of Coq's Type System Support for Extraction

Ensure that no \texttt{Set} value depends on a \texttt{Prop} value.

In a particular way that makes erasure impossible....
Preventing Bad Dependencies by Limiting Eliminations

match \( n \) with
| \( O \) => \( O \)
| \( S n' \) => \( n' \)
end

We are eliminating a value of sort \( \text{Set} \) to produce a value of sort \( \text{Set} \).
# Elimination Restrictions

<table>
<thead>
<tr>
<th>Producing a...</th>
<th>Set</th>
<th>Prop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set</strong></td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Prop</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Set -> Set example:**
Adding natural numbers

**Prop -> Set example:**
From a proof that there exists \( x \) satisfying \( P \), compute \( x \).

*Eliminate a...*

**Set -> Prop example:**
Proving properties of addition

**Prop -> Prop example:**
"If there exists \( x \) satisfying \( P \), then there exists \( y \) satisfying \( Q \)."