Verifying programs with Complex data structures using Coq

Kenneth Roe (http://www.cs.jhu.edu/~roe)
Scott F. Smith (http://www.cs.jhu.edu/~scott)
The Johns Hopkins University

Our approach, step 1: Write program and invariants

Below is shown a tree traversal program that has been converted from C to the PEDANTIC imperative language and its pre- and post condition. The user needs to create the pre- and post-conditions.

Pre-condition

Post-condition

The Advanced Rewriting Plugin

Will be used to speed up simplification in PEDANTIC

ML plugin for Coq
- Incorporates many rules for simplifying terms
- Special matching for AC functions and predicates
- Transitive closure algorithm for equality, POs and TDs
- Term representation optimizations
- Interpreting of symbols
- Bignum number representation for rats
- Interpreting of expressions

Coq’s representation of natural numbers is unary notation
- Great for logicians but no one else
- This is extremely inefficient for big numbers such as 1,000,000,000
- Hidden because Coq printouts convert to standard base 10 representation

The advanced rewrite plugin uses a more standard bignum representation

printAST 100.
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2)
(Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2)) (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2))

Step 2: top level verification

A top level prolog that propagates the invariant is created
- Much of this work can be automated
- The user needs to supply
  - while loop invariants
  - results from merging if-then-else branches

Step 3: fill in the details.

- Many bugs will be found by step 2
- Developers may opt to skip this step to save time
- Top level proof requires 12 lemmas (only one merge is non-trivial)
- 5 for checking the validity of pointer references and a delete operation
- 4 for merge steps
- 3 for entailment

Our goal: Minimize proof development time and still find the bugs
- Key measure: “formal verification time/program development time”
- A fully automated static analysis tool is not practical as it cannot fully capture the data structure invariants needed to find important bugs

A more complex case, 200 line C program implementing DPLL, invariant shown below

The design of both CoqPIE and the PEDANTIC framework come from informally working many verifications by hand. Our goal is to make formal verification more practical.

CoqPIE: An IDE for Coq
To improve proof development productivity, we developed CoqPIE. CoqPIE features
- full product management
- caching of proof states to allow quick browsing
- replay assist to help replay a proof when its statement changes

Verifying programs with Complex data structures using Coq

Kenneth Roe (http://www.cs.jhu.edu/~roe)
Scott F. Smith (http://www.cs.jhu.edu/~scott)
The Johns Hopkins University

Our approach, step 1: Write program and invariants

Below is shown a tree traversal program that has been converted from C to the PEDANTIC imperative language and its pre- and post condition. The user needs to create the pre- and post-conditions.

Pre-condition

Post-condition

The Advanced Rewriting Plugin

Will be used to speed up simplification in PEDANTIC

ML plugin for Coq
- Incorporates many rules for simplifying terms
- Special matching for AC functions and predicates
- Transitive closure algorithm for equality, POs and TDs
- Term representation optimizations
- Interpreting of symbols
- Bignum number representation for rats
- Interpreting of expressions

Coq’s representation of natural numbers is unary notation
- Great for logicians but no one else
- This is extremely inefficient for big numbers such as 1,000,000,000
- Hidden because Coq printouts convert to standard base 10 representation

The advanced rewrite plugin uses a more standard bignum representation

printAST 100.
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2)
(Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2)) (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2) (Construct (Name nat) 2))

Step 2: top level verification

A top level prolog that propagates the invariant is created
- Much of this work can be automated
- The user needs to supply
  - while loop invariants
  - results from merging if-then-else branches

Step 3: fill in the details.

- Many bugs will be found by step 2
- Developers may opt to skip this step to save time
- Top level proof requires 12 lemmas (only one merge is non-trivial)
- 5 for checking the validity of pointer references and a delete operation
- 4 for merge steps
- 3 for entailment

Our goal: Minimize proof development time and still find the bugs
- Key measure: “formal verification time/program development time”
- A fully automated static analysis tool is not practical as it cannot fully capture the data structure invariants needed to find important bugs

A more complex case, 200 line C program implementing DPLL, invariant shown below

The design of both CoqPIE and the PEDANTIC framework come from informally working many verifications by hand. Our goal is to make formal verification more practical.

CoqPIE: An IDE for Coq
To improve proof development productivity, we developed CoqPIE. CoqPIE features
- full product management
- caching of proof states to allow quick browsing
- replay assist to help replay a proof when its statement changes