We are working on a verification of this invariant.

variables.

We implemented a variation that is about 200 lines of code.

- Implementation code is small
- DPLL algorithm is a well suited example

To study data structure invariant verification, we chose to verify a simplified version of DPLL.

The object of our research is to create tools for documenting and reasoning about complex data structure invariants.

Our strategy for improving proof development productivity is to introduce a new IDE addressing these issues:

- (1) Makes unit propagation very efficient
- (2) Perform unit propagation to find additional assignments implied by the choices already made
- (3) Backtrack and change choices when a contradiction is found

Watch variables

(1) Makes unit propagation very efficient
(2) Two unassigned variables chosen at random
(3) If a watch variable in a clause is assigned, then choose a replacement. If one cannot be found, then there is only one assigned variable left and a unit propagation needs to be performed.

Blue variables in the example above are initial watch variables. After A is assigned false, we move the watch that was on A in the two clauses to C and D respectively (the brown variables).

Our C program uses the following data structures to store the clauses, the watch variable linked lists, the assignments and a “todo” queue.

The TREE declarations above define linked lists. They do not define the back pointers and assignment array v3 are consistent. We use (a,b)--->c as an abbreviation for (a,b)--->c.

The TREE(assignments_to_do_head,v1,sizeof_assignment_stack,[next_offset]) *


This code removes the most recent assignment. Proving that the invariant above is correct for each clause involved the following cases for each clause:

1) Two watch variables are assigned before.
2) All but one variable is assigned but the assignment removed does not appear in the clause.
3) All but one variable is assigned and the assignment removed does appear in the clause.
4) At least one of the assignments satisfies the clause.
5) At least one of the assignments satisfies the clause and the one and only satisfying assignment is the variable being removed.
6) At least one of the assignments satisfies the clause. The one assignment removed is not a satisfying assignment.
7) At least two of the assignments satisfies the clause. The one assignment removed is a satisfying assignment.

The proof that the invariant holds after this code took over 2000 lines of Coq proof script code.

Coq data structure invariant

Contains all of the important properties in about 50 lines of Coq code. A fragment of the invariant is shown in the box below. Here is an informal statement of the watch variable invariant:

All clauses have two watch variables. For each clause, one of the following three cases is true:

1) The two watch variables are unassigned.
2) All but one variable is assigned in the clause. One of the watch variables is the unassigned variable. The other is the most recently assigned variable.
3) At least one of the assignments satisfies the clause. If one or both watch variables are assigned, then those assignments were either a satisfying assignment or done after the first satisfying assignment.

Consider this piece of code that removes the most recent assignment:

This code removes the most recent assignment. Proving that the invariant above is correct for each clause involved the following cases for each clause:

1) Two watch variables are assigned before.
2) All but one variable is assigned but the assignment removed does not appear in the clause.
3) All but one variable is assigned and the assignment removed does appear in the clause.
4) At least one of the assignments satisfies the clause.
5) At least one of the assignments satisfies the clause and the one and only satisfying assignment is the variable being removed.
6) At least one of the assignments satisfies the clause. The one assignment removed is not a satisfying assignment.
7) At least two of the assignments satisfies the clause. The one assignment removed is a satisfying assignment.

The proof that the invariant holds after this code took over 2000 lines of Coq proof script code.

The Challenge of Prover Productivity

- Prover productivity ratio
  - Time to verify code
  - Time to develop code
- Currently, this ratio is well over 100/1 for any interactive theorem based verification
- At 10/1, a fairly valuable software development tool could be produced

The DPLL algorithm

Efficient SAT solving algorithm for CNF expressions such as:

\[(A_1 \lor \cdots \lor A_n) (B_1 \land \cdots \land B_m)\]

1 byte

We suggest two approaches as to how our techniques could be used to block Heartbleed:

(1) Since packets being received cannot be trusted, add sanity checks to verify that once the invariants are satisfied, that unauthorized information cannot leak out
(2) CoqPIE provides tactics to break up large proofs into lemmas–this often improves the performance of Coq
(3) CoqPIE will replace a proof script with admit if you are simply jumping over an entire theorem

Preventing Heartbleeds

Heartbleed is a bug in OpenSSL code
- Heartbleed is sent once every two seconds by one
- A Heartbleed sent once every two seconds by one
- The other side receive the message extracting the payload data
- Exploitation of the bug is shown below
- Message has broken payload_len field
- payload data
- No bounds check on memory
- Data beyond end of allocated block copied over

CoqPIE: Improving proof development productivity

Our strategy for improving proof development productivity is to introduce a new IDE addressing these issues:

(1) Often it is useful to quickly review earlier goal hypothesis states. Existing IDEs require the Coq prove itself to backup or move forward in a script. On a complex proof each step can take 30 seconds to backup or reevaluate.
(2) Ltac--the scripting language for Coq has many holes. Many simple rules cannot be expressed
(3) Navigating to specific definition declarations can be difficult if there is a large amount of proof script code.

CoqPIE is about 13000 lines of Python source code

Windowing implemented with Tkinter

The one assignment removed is not a satisfying assignment.

Dependency information

When a definition or lemma is edited, it can impact the validity of other declarations that depend on it.

- CoqPIE automatically marks declarations that have been invalidated.

Mitigating Coq performance

One of the biggest sources of productivity problems is the speed of the Coq theorem prover. Complex proofs can take hours (or even days) to fully verify. A single step can take a minute to process in a long proof.

(1) Intermediate goals are cached. Simply reviewing a proof does not invoke Coq.
(2) CoqPIE provides tactics to break up large proofs into lemmas–this often improves the performance of Coq
(3) CoqPIE will replace a proof script with admit if you are simply jumping over an entire theorem

Replay

Often the process of developing a theorem reveals errors in the statement of a theorem. When that statement is changed, the script needs to be adapted. Changes may involve the following:

(1) Removing proofs for subgoals that vanish
(2) Creating stubs for new subgoals
(3) Removing the error discovered in a theorem declaration

Key Features of CoqPIE

- CoqPIE saves Coq output after each proof step
- CoqPIE maintains an AST parse tree of the source code and Coq output
- AST incrementally updated as edits are made
- The relationship between AST nodes and source text is maintained.
- Complex tactics such as replay and lemma extraction built on top of AST representation
- CoqPIE manages the entire project—not just one file
- Treatview shows summary of all files and definitions
- Definitions and proofs that need to be recomplied due to changes to dependent definitions are highlighted
- Difference highlighting allows one to quickly see changes after each proof step

CoqDPLL Invariant fragment