Scaling Coq based separation logic to more complex problems

Kenneth Roe
The Johns Hopkins University
July 2017

Motivation

- Most real world bugs can be traced to data structure invariant violations
  - Consider Heart Bleed
- $100,000,000 business opportunity for an effective tool that documents and verifies invariants
  - Existing tools such as Coverity don't provide a rich enough verification
- The key issue is improving proof development productivity

DPLL verification

- What is DPLL
  - A SAT solver algorithm that uses many complex data structures
- Why DPLL
  - Relatively small program
  - Data structures complex enough to expose many important issues in verifying large programs
- Size of algorithm (my simplified implementation)
  - 200 lines of C code
  - 100 line Coq invariant

PEDANTIC Framework

- Coq base separation logic
  - Deep model
  - Structured proof development
    - Top level proof always follows structure of the program and can be mostly generated automatically
  - Simplification

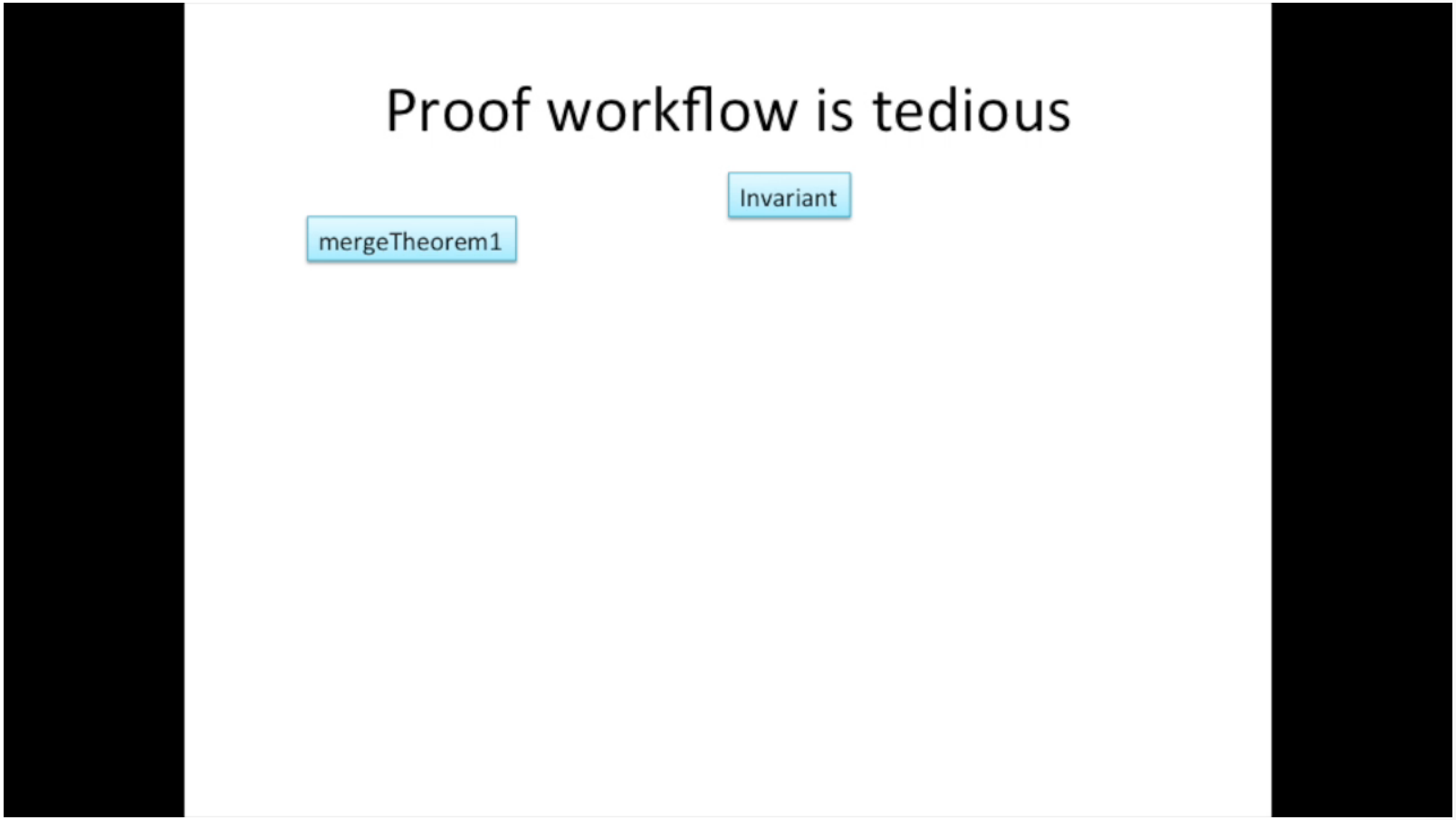
DPLL invariant

```
Definition invariant: absStateBasic :=
  (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT (
    TREE(!!clauses,v(0),#sizeof_clause,(#next_offset::nil)) **
    TREE(!!assignments_to_do_head,v(1),#sizeof_assignment_stack,
(#next_offset::nil)) **
    TREE(!!stack,v(2),#sizeof_assignment_stack,(#next_offset::nil)) **
    ARRAY(!!assignments,#var_count,v(3)) **
    ARRAY(!!watches,#var_count,v(4)) **
    (* Assertions that the stack and assignments array contain the same set
      of assignments *)
    (AbsAll TreeRecords(v(2))
      ([--(v(2),v(5))-->stack_var_offset <<<< #var_count] **
       ([--(v(2),v(5))-->stack_val_offset ==== #1] *\/* [--(v(2),v(5))--
>stack_val_offset ==== #2]) **
       ([nth(v(3),--(v(2),v(5))-->stack_var_offset)=====--(v(2),v(5))--
>stack_val_offset]) **
       (AbsAll TreeRecords(nth(find(v(2),v(5)),#2))
          ([~~(--(v(2),v(5))-->stack_var_offset====--(nth(find(v(2),v(5)),#2),v(6))--
>stack_val_offset)]))))) **
    (AbsAll range(#0,#(var_count))
      ([nth(v(3),v(5))====#0] *\/*
       AbsExists (TreeRecords(v(2)))
          ([[--(v(2),v(6))-->stack_var_offset====v(5) //\\
           --(v(2),v(6))-->stack_val_offset====nth(v(3),v(5))]]) )) **
    (* Assertion defining the prev pointer in the assignments_to_do
      doubly linked list *)
    (AbsAll TreeRecords(v(1))
      ([(--(v(1),v(5))-->prev_offset====#0 //\\ (!!
assignments_to_do_head)====v(5)) \\//
       (--(v(1),v(5))-->prev_offset inTree v(1) //\\
        --(v(1),--(v(1),v(5))-->prev_offset)-->next_offset====v(5))])) **
    (AbsEach range(#0,#(var_count))
      (* Define the basic linked list connecting the watch variables
        inside the clauses linked list *)
      (AbsExistsT
        ((Path((nth(v(4),v(5))), v(0), v(6), #sizeof_clause, ((#watch_next_offset+
+++v(5))::nil))) **
          (* Define the prev variable and the fact that if null we are at
            the head of the list *)
          (AbsAll TreeRecords(v(6))
            ([(--(v(6),v(7))-->(#watch_prev_offset++++v(5))====#0 //\\
nth(v(4),v(5))====v(6)) \\//
              (--(v(6),--(v(6),v(7))-->(#watch_prev_offset++++v(5)))---
>(#watch_next_offset++++v(5)))===v(7)])))))) **
```

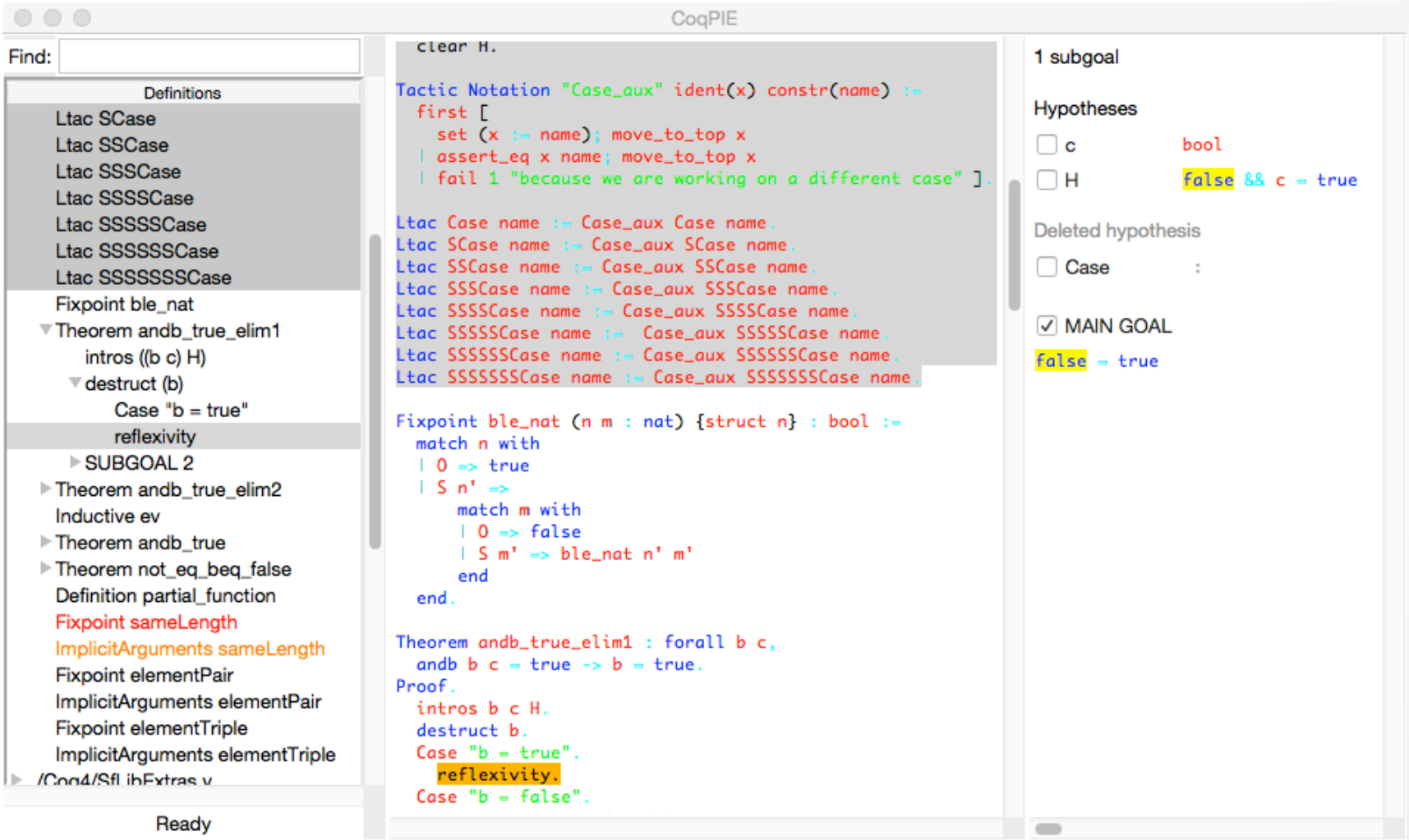
```
(AbsAll TreeRecords(v(0))
  (* The current assignment is consistent with the clause *)
  ((AbsExists range(#0,#(var_count))
     ([(--(v(0),v(5))--->(#positive_lit_offset++++v(6)) //\\
       (nth(v(3),v(6))====#2 \\// nth(v(3),v(6))====#0)) \\//
      (--(v(0),v(5))--->(#negative_lit_offset++++v(6)) //\\
       (nth(v(3),v(6))====#1 \\// nth(v(3),v(6))====#0)])])) **
  (*
   * make sure that if the watch_var field is non-zero (pointing to
   * a variable) that watch_next and watch_prev put this clause into
   * the linked list for the watch variable.
   * Also, for all watch variables, either positive_lit or negative_lit
   * is true.
   *)
  (AbsAll range(#0,#(var_count))
    ([
     (--(v(0),v(5))--->(#watch_var_offset++++v(6))====#0) \\//
      (--(v(0),v(5))--->(#positive_lit_offset++++v(6))) \\//
      (--(v(0),v(5))--->(#negative_lit_offset++++v(6)))])) **
  (AbsAll range(#0,#(var_count))
    ([
     (~~(--(v(0),v(5))--->(#watch_var_offset++++v(6))====#0) //\\
     (~~(--(v(0),v(5))--->(#watch_prev_offset++++v(6))====#0) \\//
       nth(v(4),v(6))====v(5))) \\//
     (--(v(0),v(5))--->(#watch_var_offset++++v(6))====#0 //\\
      --(v(0),v(5))--->(#watch_prev_offset++++v(6))====#0 //\\
      ~~(nth(v(4),v(6))====v(5))])) **
  (* Make sure there are precisely two watch variables per clause or all
variables are watches,
     needs fixing? *)
  (SUM(range(#0,#(var_count)),ite((--(v(0),v(5))--->(#watch_var_offset+++
+v(6))),(#1),(#0)),#2)) **
  (* Watch variable invariant--case 1:  All but one variable in the
    clause are assigned, any watch variable pointing to an assigned
    variable is pointing to a variable that was assigned after all
    other assigned variables in the clause.  Also, one of the two
    watch variables points to the one unassigned variable *)
  ((((SUM(range(#0,#(var_count)),
    (((--(v(0),v(5))--->(#positive_lit_offset++++v(6))) \\// (--(v(0),v(5))---
>(#negative_lit_offset++++v(6)))) //\\
       ite(nth(v(3),v(6))====#0,#1,#0)),
    #1) **
  (* The one unassigned literal is a watch--needs fixing? *)
  (AbsAll range(#0,#(var_count))
     ([(#0<<<<--(v(0),v(5))--->(#watch_var_offset++++v(6)) //\\
       (nth(v(3),v(6))====#0)) \\//
       (
        ((#0<<<<nth(v(3),v(6))) \\// ((--(v(0),v(5))--->(#positive_lit_offset+++
+v(6))====#0 //\\
          --(v(0),v(5))--->(#negative_lit_offset++++v(6))====#0))))])) **
  (AbsAll range(#0,#(var_count))
    (AbsAll range(#0,#(var_count))
      (([--(v(0),v(5))--->(#watch_var_offset++++v(6)) \\//
        ((((--(v(0),v(5))--->(#positive_lit_offset++++v(6)))====#0 //\\ (--
(v(0),v(5))--->(#negative_lit_offset++++v(6))====#0)))) \\//
         ~~(--(v(0),v(5))--->(#watch_var_offset++++v(7))) \\//
        nth(v(3),v(6))====#0 \\// nth(v(3),v(7))====#0 \\// v(6)====v(7)])
*\/*
        (AbsExists TreeRecords(v(2))
          (([--(v(2),v(8))-->stack_var_offset====v(7)]) **
           (AbsExists TreeRecords(find(v(2),v(8)))
             ([--(v(2),v(9))-->stack_var_offset====v(6)])))))))))) *\/*
    (* Watch variable invariant case 2: One of the assignments already
      satisfies the clause, if a watch variable is assigned a value,
      then that value must be a satisfying assignment or occured
      after a satisfying assignment *)
    ( (AbsExists range(#0,#(var_count))
       ([(--(v(0),v(5))--->(#positive_lit_offset++++v(6)) //\\
nth(v(3),v(6))====#2) \\//
        (--(v(0),v(5))--->(#negative_lit_offset++++v(6)) //\\
nth(v(3),v(6))====#1)])) **
      (AbsAll range(#0,#(var_count))
       ((([#0====nth(v(3),v(6))])) *\/*
        ([--(v(0),v(5))--->(#watch_var_offset++++v(6))====#0]) **
        ([#0<<<<nth(v(3),v(6))])) *\/*
        (AbsExists TreeRecords(v(2))
         (([--(v(2),v(7))-->stack_var_offset====v(6)]) **
          (AbsExists TreeRecords(find(v(2),v(7)))
           ([((#0 <<<< (--(v(0),v(5))--->(#positive_lit_offset++++ --(v(2),v(8))--
>stack_var_offset))) //\\
             --(v(2),v(8))-->stack_val_offset====#2) \\//
            ((#0 <<<< --(v(0),v(5))--->(#negative_lit_offset++++ --(v(2),v(8))--
>stack_var_offset)) //\\
             --(v(2),v(8))-->stack_val_offset====#1)])))))) *\/*

    (* Watch variable invariant case 3: both watch variables point to
      unassigned variables *)
    (AbsAll range(#0,#(var_count))
      ([--(v(0),v(5))--->(#watch_var_offset++++v(6))====#0 \\//
nth(v(3),v(6))====#0]))
)))))))).
```

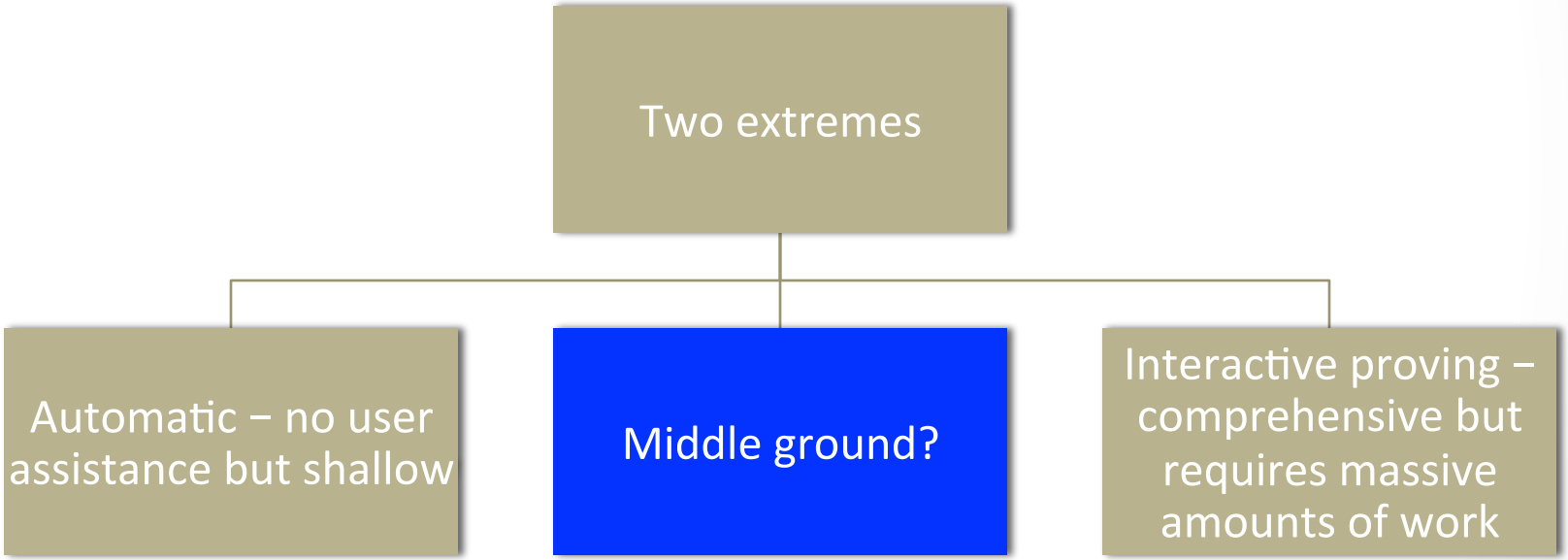
A short video



CoqPIE


CoqPIE
Find:
Definitions
Ltac SCase
Ltac SSCase
Ltac SSSCase
Ltac SSSSCase
Ltac SSSSSCase
Ltac SSSSSSCase
Ltac SSSSSSSCase
Fixpoint ble_nat
Theorem andb_true_elim1
intros ((b c) H)
destruct (b)
Case "b = true"
reflexivity
SUBGOAL 2
Theorem andb_true_elim2
Inductive ev
Theorem andb_true
Theorem not_eq_beq_false
Definition partial_function
Fixpoint sameLength
ImplicitArguments sameLength
Fixpoint elementPair
ImplicitArguments elementPair
Fixpoint elementTriple
ImplicitArguments elementTriple
Ready
clear H.
Tactic Notation "Case_aux" ident(x) constr(name) :=
first [
set (x := name); move_to_top x
| assert_eq x name; move_to_top x
| fail 1 "because we are working on a different case"].
Ltac Case name := Case_aux Case name.
Ltac SCase name := Case_aux SCase name.
Ltac SSCase name := Case_aux SSCase name.
Ltac SSSCase name := Case_aux SSSCase name.
Ltac SSSSCase name := Case_aux SSSSCase name.
Ltac SSSSSCase name := Case_aux SSSSSCase name.
Ltac SSSSSSCase name := Case_aux SSSSSSCase name.
Ltac SSSSSSSCase name := Case_aux SSSSSSSCase name.
Fixpoint ble_nat (n m : nat) {struct n} : bool :=
match n with
| O => true
| S n' =>
match m with
| O => false
| S m' => ble_nat n' m'
end
end.
Theorem andb_true_elim1 : forall b c,
andb b c = true -> b = true.
Proof.
intros b c H.
destruct b.
Case "b = true".
reflexivity.
Case "b = false".
1 subgoal
Hypotheses
c bool
H false && c = true
Deleted hypothesis
Case :
MAIN GOAL
false = true

A Verification middle ground



Verification of traversal (portion)

…

```
(* IF (ALand (!Tmp_l === A0) (!Tmp_r === A0)) *)
eapply if_statement. simpl.

  (* IF (!I === A0) *)
  eapply if_statement. simpl.

    (* T ::= A0 *)
    pcrunch.

  (* ELSE *)

    (* CLoad T (!I)++A1 *)
    eapply compose. pcrunch.

    (* CLoad Tmp_l (!I)++A0 *)
    eapply compose. pcrunch.

    (* DELETE !I, A2 *)
    eapply compose.
    Set Printing Depth 200. pcrunch.
    eapply deleteExists1. apply H0.

    (* I ::= !Tmp_l *)
    pcrunch.

    pcrunch. pcrunch. pcrunch. pcrunch.

  (* FI *)
  apply mergeTheorem1.
```

…

Verification of traversal

- Main proof requires 12 lemmas
  - Main lemma specifies abstract state at each line
  - Many can be done automatically
  - We can skip lemmas for light weight verificaiton

Conclusion

- DPLL well under way
  - Many scaling issues have been addressed
- For more information
  - http://www.cs.jhu.edu/~roe
  - Kenneth Roe and Scott Smith, “Using the Coq Theorem Prover to Verify Complex Data Structure Invariants”, *MEMOCODE 2017*
  - Kenneth Roe and Scott Smith, “CoqPIE: An IDE aimed at improving proof development productivity (rough diamond)”, *Interactive Theorem Proving*, 2016