Hartford Coliseum Collapse (1978)

CAD software used to design the Coliseum contained a bug that lead to incorrect design decisions

www.devtopics.com/20-famous-software-disasters

Refinements to techniques for verifying shape analysis invariants in Coq

Kenneth Roe http://www.cs.jhu.edu/~roe The Johns Hopkins University

Ariane 5 Flight 501 (1996)

Destructed 40 seconds after takeoff due to a bug in the on-board guidance software

Wikipedia

Summary

Formal Methods for Imperative Languages Such as C

Objective: Develop a Framework in Coq capable of reasoning about realistic programs and the complex data structures they manipulate

Research Contributions

Extension of Coq separation logic reasoning to larger programs with more complex data structures.

Creation of a library of useful predicates, functions and tactics
- Deep model allows greater control over the design of tactics

Key challenge: Performance tuning
- Tradeoffs between performance and automation

Development of a powerful simplification tactic
- Simplification tactic executed after every major proof step
- Based on term rewriting (with contextual rewriting) concepts
- Automates reasoning about associativity, communitivity and other simple property classes
- Design decisions in creating canonical form addressed

Timeline of related work:

- 1977 — Abstract Interpretation, Cousot & Cousot
- 1998 — Early shape analysis work, Reps & Sagiv
- 2002 — Separation Logic, Reynolds, O'Hearn & Yang
- 2006 — Separation Logic Applications, O'Hearn, Yang, Cook, Distefano
- 2009 — Coq separation logic tactics, McCreight
- 2010 — YNot, Malecha, Morrisett, Chlipala
- 2011 — Bedrock, Chlipala
- 2012 — Charge!, Bengston & Birkedal

Separation Logic basics

The following constructs come from Bedrock/YNot/Charge!:
(1) Higher order separation * operator
(2) Lifted AbsAll, AbsExists
(3) [P] - a separation logic predicate that asserts P and that the heap is empty

We have added the following:
(1) TREE is a generic recursive predicate for constructing linked lists and trees in the heap.
(2) Path (See the DPLL example) is like TREE but it creates a list or tree structure embedded inside another list or tree.
(3) (SUM x in R, e)==total
is used to sum up values over a range
(4) Foreach x in R, P.
P is evaluated for all values of x in R. The results are composed together using the separation *
(5) *\/* - separation "or"
(6) library of useful functions for separation predicates

Property classes

Soundness of invariants
Termination or completeness proofs not produced

Result summary

The work on the examples so far indicate that the approach is promising.

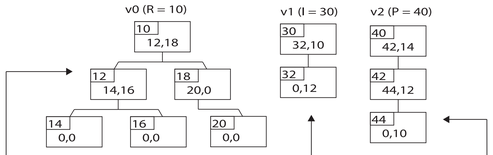
Proof size, invariant size, and Coq verification times are all reasonable.

Verifying a small tree traversal program

The main function, pre_order, takes as input a pointer to a tree structure and generates a list which represents a traversal of the tree.

```
struct list    {                          t = NULL;
  struct list *n;                       } else {
  struct tree *t;                         struct list *tmp = i->n;
  int data;                               t = i->t;
};                                        free(l);
                                          l = tmp;
struct tree    {                        }
  struct tree *l, *r;                  } else if (t->r==NULL)      {
  int value;                           t = t->l;
};                                     } else if (t->l==NULL)      {
struct list *p;                        t = t->r;
void build _pre_order(struct tree *r)  } else {
{ struct list *i = NULL, *n, *x;       n = i;
  struct tree *t = r;                  i = malloc(
  p = NULL;                              sizeof(struct list));
  while (t) {                          i->n = n;
    n = p;                             x = t->r;
    p = malloc(sizeof(struct list));   i->t = x;
    p->t = t;                          t = t->l;
    p->n = n;                        }
    if (t->l==NULL && t->r==NULL)    }
      if (i==NULL)    {
```

Heap snapshot



Invariant

(1) There is a tree and two lists all properly formed--no cycles, no dangling points
(2) Each of the nodes in the two lists contains a pointer a node in the tree.

Coq Representation:

```
AbsExists v0.AbsExists v1.AbsExists v2.
  TREE(R, v0, 2, [0,1] )*
  TREE(I, v1, 2, [0] )*
  TREE(P, v2, 2, [0] )*
 AbsAll v3 in TreeRecords (v1).
     [nth(find (v1, v3), 2) inTree v0]*
 AbsAll v3 in TreeRecords (v2).
     [nth(find (v2, v3), 2) inTree v0]*
     [T = 0 \/ T inTree v0]
```

Post-condition

Proof excerpt

Goals are Hoare triples of form {{pre}}stmts{{post}}

Proof steps involve the following:
(1) forward propagating over unit statements
(2) applying Hoare rules to break up if-then-else, while and statement sequences
(3) folding or unfolding recursive predicates
(4) merging two states at the end of an if-then-else
(5) proving one state implies another state

```
Theorem loopInvariant : {{afterInitAssigns}}loop{{afterWhile,NoResult}}.
Proof.
(*{{AbsExistsT v0 .
   (AbsExistsT v1.
     (AbsExistsT v2.
       (TREE(R, v0, 2, [0,1]) *
        TREE(I, v1, 2, [0]) *
        TREE(P, v2, 2, [0]) *
        AbsAll v3 in TreeRecords(v1). ([nth(find(v1, v3), 2) inTree v0]) *
        AbsAll v3 in TreeRecords(v2). ([nth(find(v2, v3), 2) inTree v0]) *
        [T == 0 \/ T inTree v0])))}}
WHILE not (T == 0)
DO N := P;
  NEW P, 2;
  *(P + 1) := T;
  *P := N;
  Tmp_l := *T;
  Tmp_r := *(T + 1);
  IF (Tmp_l == 0) and (Tmp_r == 0)
  THEN IF I == 0 THEN T ::= 0
       ELSE T := *(I + 1);
            Tmp_l := *(I + A0); DELETE I, 2; I ::= Tmp_l
  ELSE IF Tmp_l == 0 THEN T ::= *(T + 1)
       ELSE IF Tmp_r == A0 THEN T ::= *T
            ELSE N := I;
                 I := NEW 2;
                 *(I + 0) := N;
                 Tmp_l := *(T + 1);
                 *(I + 1) := (!Tmp_l); T := *T;
{{AbsExistsT v0 .
   (AbsExistsT v1.
     (AbsExistsT v2.
       (TREE(R, v0, 2, [0,1]) *
        TREE(I, v1, 2, [0]) *
        TREE(P, v2, 2, [0]) *
        AbsAll v3 in TreeRecords(v2), ([nth(find(v2, v3), 2) inTree v0]) *
        [T == 0])))}}*)
```

Pre-condition

Program

Coq Verification goal

```
(* WHILE not (T == 0) DO *)
eapply strengthenPost.
eapply while with (invariant := loopInv). unfold loopInv.
eapply strengthenPost.
(* N ::= P; *)
eapply compose. (eapply basicAssign; pcrunch).

...

(* IF ((Tmp_l == 0) and (Tmp_r == 0)) *)
eapply if_statement.
(* IF (I == 0) *)
  eapply if_statement.
    (* T ::= 0 *)
    (eapply basicAssign; pcrunch).
    (* ELSE *)
    simp. simp. simp. simp.

    ...

    startMerge.
      doMergeStates. DMRemoveZeroTree2 ((!!!) : absExpBasic).
      DMRemoveZeroAll2.
      (* Special case where two predicates need to be or'ed together *)
      eapply DMOrPredicates1. instantiate (2 := (!!(T) ==== #0)).
      solveSPickElement.
      instantiate (2 := (!!(T) inTree v(3))). solveSPickElement. pcrunch.
    finishMerge.

  (* ELSE *)

    simp. simp. simp.
    (* IF Tmp_l == 0 THEN T ::= *T *)
    eapply if_statement. simpl.

    simp. simp. simp.

    (* T ::= *(T + 1) *)
    load_traverse.
    (* ELSE *)
    ....
```

Forward Propagations

State Merging

Forward Propagation

Explanation of Coq representation

The three TREE assertions characterize the tree and the two lists of the heap. Parameters:
(1) the root of the tree
(2) a variable holding an equivalent functional representation (discussed below);
(3) the size of a record in words;
(4) and, a list of offsets for all the child node pointers. The tree needs two such offsets, [0,1] and the lists only need one, [0].

Two AbsAll clauses assert for each list that the pointers in the second field of each node in the list point into the tree.

Functional Equivalent Representation

Characterize data fields saved, structure of the tree or list and the locations of records on the heap.

Represented by lists whose elements are either naturals or (recursively) lists of lists or naturals.

For the snapshot from the picture above, v0 would for example be the list
[10,([12,[14,[0],[0]],[16,[0],[0]]]),([18,[20,[0],[0]],[0]])]

Other Constructs

(1) TreeRecords is a function that returns the list of record pointers given a functional representation.
* TreeRecords([12,[14,[0],[0]],[16,[0],[0]]]) returns [12,14,16].
(2) "x inTree y" is shorthand for "x in TreeRecords(y)"
(3) find takes an address in a tree and a tree functional representation, and returns the subtree rooted at that address. find(14,[12,[14,[0],[0]],[16,[0],[0]]]) returns [14,[0],[0]].

Statistics

Code size: ~30 lines
Invariant size: ~10 lines
Proof check time: ~5 minutes
Main proof size: ~220 lines

Status

Top level proof complete
Many lemmas need to be proven

Verifying the DPLL algorithm

Efficient SAT solving algorithm for CNF expressions such as:
(A v ~B v C) ^
(A v ~C v D)

DPLL algorithm:
(1) Choose a variable and assign it a value
(2) Perform unit propagation to find additional assignments by the choices already made.
(3) Backtrack and change choices when a contradiction is found

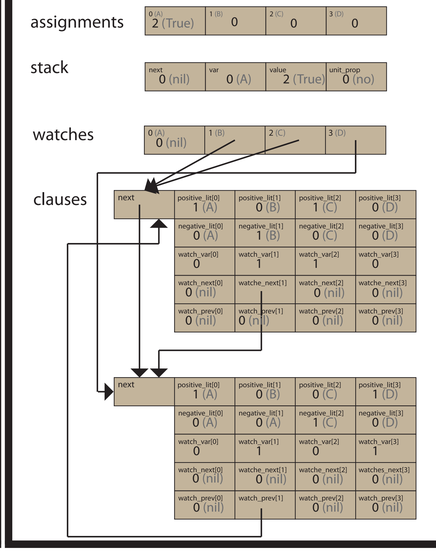
Watch variables
(1) Makes unit propagation very efficient
(2) Two unassigned variables chosen at random
(3) If a watch variable in a clause is assigned, then choose a replacement. If one cannot be found, then there is only one assigned variable left and a unit propagation needs to be performed.

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.:

```
#define VAR_COUNT 4

char assignments[VAR_COUNT];

struct clause {
  struct clause *next;
  char positive_lit[VAR_COUNT];
  char negative_lit[VAR_COUNT];
  char watch_var[VAR_COUNT];
  struct clause *watch_next[VAR_COUNT];
  struct clause *watch_prev[VAR_COUNT];
} *clauses;

struct clause *watches[VAR_COUNT];

struct assignments_to_do {
  struct assignments_to_do *next, *prev;
  int var;
  char value;
  int unit_prop;
} *assignments_to_do_head, *assignments_to_do_tail;

struct assignment_stack {
  struct assignment_stack *next;
  int var;
  char value;
  char unit_prop;
} *stack;
```

Here is a diagram showing what the data structures look like right after A is assigned false.



Coq data structure invariant

Contains all of the important properties in about 30 lines.

The first part of the invariant are spacial constructs asserting the two arrays and three dynamic data structures in the heap. ARRAY(root, count, functional_representation) is a spatial predicate for arrays. The functional representation is a list of the elements.

```
AbsExistsT v0 . AbsExistsT v1 . AbsExists v2 . AbsExistsT v3. AbsExistsT v4.
    TREE(clauses,v0,sizeof_clause,[next_offset]) *
    TREE(assignments_to_do_head,v1,sizeof_assignment_stack,[next_offset]) *
    TREE(stack,v2,sizeof_assignment_stack,[next_offset]) *
    ARRAY(assignments,var_count,v3) * ARRAY(watches,var_count,v4) *
```

Next, we add on two assertions that guarantee that both the assignment_stack v2 and assignment array v3 are consistent. We use (a,b)--->c as an abbreviation for nth(find(a,b),c).

```
    (AbsAll v5 in TreeRecords(v2) . ([nth(v3,(v2,v5)--->stack_var_offset)==(v2,v5)--->stack_val_offset])) *
    (AbsAll v5 in range(0,var_count-1) . ([nth(v3,v5)==0] *\/* AbsExists v6 in (TreeRecords(v2)) .
          ([((v2,v6)--->stack_var_offset==v5 /\ (v2,v6)--->stack_val_offset==nth(v3,v5))])) ) *
```

The TREE declarations above define linked lists. They do not define the back pointers of a double linked list to do this for the assignments_to_do_head list, we add the following assertion:

```
    (AbsAll v5 in TreeRecords(v1) .
        ([((v5,v1)--->prev_offset==0 /\ (assignments_to_do_head)==v5) \/
         ((v5,v1)--->prev_offset inTree v1 /\ (v5,(v5,v1)--->prev_offset)--->next_offset==v5)])) *
```

Now we define the linked lists for each of the watch variables. Path is like TREE but it defines lists inside of other structures. It is used for the embedded watch variable linked lists. It takes the form:
Path(root, parent_functional_data,child_functional_form,node_size,pointer_offsets)
We also put in the assertion for the prev links:

```
    (AbsEach v6 in range(0,var_count-1)
      (Path((nth(v4,v5)), v0, v6, sizeof_clause, [watch_next_offset+v5]) *
       (AbsAll v7 in TreeRecords(v6)
         ([((v6,v7)--->(watch_prev_offset+v5)===0 /\ nth(v4,v5)====v6) \/
          ((v6,(v6,v7)--->(watch_prev_offset+v5))--->(watch_next_offset+v5))==v7)])) *
```

Next, we have an assertion to ensure that our current variable choices are consistent with each clause (meaning that there is still a way of doing the remaining variable assignments to satisfy each clause.

```
  (AbsEach v5 in TreeRecords(v0) (AbsExists v6 in range(0,var_count-1)
       ([((v0,v5)--->(positive_lit_offset+v6) /\ (nth(v3,v6)==2) \/ nth(v3,v6)==0)) \/
        ((v0,v5)--->(negative_lit_offset+v6) /\ (nth(v3,v6)==1 \/ nth(v3,v6)==0))])) *
```

For each position in the watch_var, watch_prev, and watch_next arrays, watch_var is non-zero if and only if watch_prev and watch_next add that variable to the corresponding watch variable linked list.

```
  (AbsAll v6 in range(0,var_count-1)
      ([(not((v0,v5)--->(watch_var_offset+v6)==0) /\
         (not((v0,v5)--->(watch_prev_offset+v6))==0) \/ nth(v4,v6)==v5)) \/
        ((v0,v5)--->(watch_var_offset+v6)==0 /\
         (v0,v5)--->(watch_prev_offset+v6)==0 /\ not(nth(v4,v6)==v5))])) *
```

Each clause has precisely two watch variables (or that all variables are watches. (ite=if-then-else)

```
    (SUM(v6 in range(0,var_count-1),ite(((v5,v0)--->(watch_var_offset+v6)),1,0)==2 *\/*
     (AbsAll v6 in range(0,var_count-1) ([not((v5,v0)--->negative_lit_offset+v6 \/ (v5,v0)->positive_lit_offset+v6) \/
                                         ((v5,v0)--->(watch_var_offset+v6))])) ) *
```

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(v6 in range(0,var_count-1),
      (((v(0),v(5))--->(positive_lit_offset+v6)) /\
        ite(nth(v3,v6)==1,1,0))+
      (((v0,v(5))--->(negative_lit_offset+v6)) /\ ite(nth(v3,v6)==2,1,0)))==
    (var_count-1) *
  (* The one unassigned literal is a watch *)
  (AbsAll v6 in range(0,var_count-1)
      ((0 < (v0,v5)--->(watch_var_offset+v6) \/
        nth(v3,v6)==0)) *
  (* and make sure that one of the watch variables is indeed the last variable on the assignment stack *)
  (AbsAll v6 in range(0,var_count-1) (AbsAll v7 in range(0,var_count-1)
        ([((v0,v5)--->(watch_var_offset+v6)) /\ not((v0,v5)--->(watch_var_offset+v7)) \/
          nth(v3,v6)==0 \/ nth(v3,v7)==0 \/ v6==v7]) *\/*
        (AbsExists v8 in TreeRecords(v2) . ([(v2,v8)--->stack_var_offset===v7]) *
          (AbsExists v9 in TreeRecords(find(v2,v8)) . ([(v2,v9)--->stack_var_offset==v6])))))))) *\/*
```

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. This allows us to get back to a consistent state as variables are unassigned one by one (in reverse order) when backtracking.

```
( (AbsExists v6 in range(0,var_count-1)
   ([((v0,v5)--->(positive_lit_offset+v6) /\ nth(v3,v6)==2) \/
     ((v0,v5)--->(negative_lit_offset+v6) /\ nth(v3,v6)==1)])) *
  (AbsAll v6 in range(0,var_count-1) ([((v0,v5)--->(watch_var_offset+v6)==0)] *\/*
    (AbsExists v7 in TreeRecords(v2) ([((v2,v7)--->stack_var_offset==v6)]) *
      (AbsExists v8 in find(v2,v7)
        ((((0 < ((v0,v5)--->(positive_lit_offset+ (v2,v8)--->stack_var_offset))) /\
           (v2,v8)--->stack_val_offset==2) \/
          ((0 < (v0,v5)--->(negative_lit_offset+ (v2,v8)--->stack_var_offset)) /\
           (v2,v8)--->stack_val_offset==1))))))) *\/*
```

Watch variable invariant case 3: both watch variables point to unassigned variables

```
  (AbsAll v6 in range(0,var_count-1)
      ([((v0,v5)--->(watch_var_offset+v6)==0 \/ nth(v3,v6)==0)])))))
```

Statistics:

Code size: ~200 lines
Invariant size: ~52 lines

Status

Top level proof started but incomplete

Acknowledgements: The author would like to thank Scott Smith, Mike Hicks and Gregory Melacha for their feedback on this poster.