Using the Coq theorem prover to verify complex data structure invariants

Kenneth Roe
The Johns Hopkins University
Baltimore, MD
roe@cs.jhu.edu

Scott F. Smith
The Johns Hopkins University
Baltimore, MD
scott@jhu.edu

ABSTRACT

While automated static analysis tools can find many useful software bugs, there are still bugs that are beyond the reach of these tools. Most large software systems have complex data structures with complex invariants, and many bugs can be traced to code that does not maintain these invariants. These invariants cannot be easily inferred by automated tools. One must use an interactive system in which developers first enter these invariants to document their software and then use a theorem prover to verify their correctness.

We describe the PEDANTIC framework for verifying the correctness of C-like programs using the Coq theorem prover. PEDANTIC is designed to prove invariants over complex dynamic data structures such as inter-referencing trees and linked lists. The language for the invariants is based on separation logic. The PEDANTIC tactic library has been constructed to allow program verifications to be done with reasonably compact proofs.

We have completed the verification of a tree traversal program. We are currently working on the verification of a C implementation of the DPLL algorithm in order to demonstrate the utility of the framework on more complex programs. Verifying programs using an interactive theorem prover is quite tedious. We discuss work being done to improve proof development productivity.

1 INTRODUCTION

Many software bugs can be traced to data structure invariant violations, some of which are quite complex. For example, a SAT solver algorithm may use both a stack (represented as a linked list) and an array to represent variable assignments. The stack allows the program to efficiently remove assignments in the reverse order that they were created, and the array allows the program to quickly find the assignment for a particular variable.

Many efforts in the research literature aim at creating fully automated static analysis tools, including [29, 26, 22]. Each tool while being automatic has substantial limitations. [29] only covers a subset of separation logic and not the separation logic augmented with invariants covered in this paper. [26] cannot easily verify invariants with non-linear equations as underlying SMT solvers are limited in this area. [22]'s algorithm only works on a subset of the examples presented in the paper. Finding data structure invariants is a difficult task and hence no program will be able to automatically find and verify all of them. Hence, we focus on creating a tool that allows the user to first document data structure invariants and then verify them.

This paper presents first steps towards a framework for verifying programs in imperative languages such as C that use complex heap based data structures. Our extensible PEDANTIC[1] framework is implemented in Coq and is based on ideas from many other separation logic frameworks such as [8, 4, 20]. Coq allows almost any invariant to be verified as critical lemmas are interactively proven. The framework implements a logic language based on separation logic for describing program invariants, and a set of tactics to propagate these assertions through the program.

One of the key challenges of designing this system is to find a good division of labor between the user and the tool. The tool obviously cannot automate everything. However, if it automates too little, then the verification process will be too tedious and developers will not use the tool. As part of our effort, we have developed a new IDE for Coq, CoqPIE, aimed at optimizing the effort needed for complex proof development tasks. This project is described in [32]. This paper concentrates on the Framework we developed in Coq for verifying data structure invariants. In our design, data structure invariants need to be entered by the user. Our experience (both from the example presented as well as a DPLL verification presently being done) shows that the size of these invariants tend not to be larger than the size of the program being verified. One key difficulty in many formal systems is inferring loop invariants. Our experience has shown that loop invariants are usually very similar to the pre- and post- condition invariants. Since the task of automatically generating a loop invariant is difficult, we require the user to enter all loop invariants.

Contributions of PEDANTIC

In order to create an easy to use yet extensible framework, we took the work of [8, 4, 20, 25, 3] and introduced enhancements. We discuss four areas of development: (1) use of a deep model; (2) use of pointers in the functional representation of a data structure; (3) introduction of a `merge` tactic to merge together branches after an if-then-else construct; and (4) use of simplification after other operations to put assertions into a canonical form.

Deep model A deep model means that rather than representation our invariants directly as propositions in the Coq theorem prover, we created a data structure which is the AST for the invariant language. We then created a number of Gallina[2] functions to manipulate these data structures. This gives us greater flexibility in

MEMOCODE '17, Vienna, Austria
© 2017 Copyright held by the owner/author(s). 978-1-4503-5093-8/17/09…$15.00
DOI: 10.1145/3127041.3127061

[1]Proof Engine for Deductive Automation using Non-deterministic Traversal of Instruction Code
[2]The functional language used by Coq

```
struct list {
 struct list *n;
 struct tree *t;
 int data;
};
struct tree {
 struct tree *l, *r;
 int value;
};
struct list *p;
void build_pre_order(struct tree *r) {
ℓ1  struct list *i = NULL, *n, *x;
ℓ2  struct tree *t = r;
ℓ3  p = NULL;
ℓ4  while (t) {
ℓ5    n = p;
ℓ6    p = malloc(sizeof(struct list));
ℓ7    p->t = t;
ℓ8    p->n = n;
ℓ9    if (t->l==NULL && t->r==NULL) {
ℓ10     if (i==NULL) {
ℓ11       t = NULL;
ℓ12     } else {
ℓ13       struct list *tmp = i->n;
ℓ14       t = i->t;
ℓ15       free(l);
ℓ16       i = tmp;
ℓ17     }
ℓ18   } else if (t->r==NULL) {
ℓ19     t = t->l;
ℓ20   } else if (t->l==NULL) {
ℓ21     t = t->r;
ℓ22   } else {
ℓ23     n = i;
ℓ24     i = malloc(
          sizeof(struct list));
ℓ25     i->n = n;
ℓ26     x = t->r;
ℓ27     i->t = x;
ℓ28     t = t->l;
ℓ29   }
ℓ30  }
ℓ31 }
```

Figure 1: Program which builds a linked list of nodes in a tree

designing tactics. It also separates the design of the algorithm from the verification of the tactic. Once a correctness theorem for a tactic is created in the framework, it never needs to be done in the verification.

Embedding pointers in a functional representation The Btree example in [20] illustrates how embedding pointers in a functional representation can be used to simplify the expression of invariants. We expand on this technique by showing how it can be used to represent cross referencing relationships from one data structure to another.

Merging and pairing At the end of an if-then-else block, our forward propagation tactics will have produced two distinct state assertions. As these assertions are derived from the same starting point, they will be similar but not the same. We have developed a `merge` tactic which works by first pairing off components in the states that are the same, and then proceeding to merge other components through more complex operations. In this process it may be necessary to invoke `fold` or `unfold` tactics to align recursive data structures to facilitate the merge. We also use this pairing technique for proving assertion entailment properties.

Simplification Often the results of forward propagating assertions through a program (or performing other operations) produces fairly complicated state assertions. PEDANTIC introduces a fairly sophisticated simplification algorithm to reduce these assertions to a more canonical form.

```
Theorem loopInvariant :
    {{afterInitAssigns}}loop{{afterWhile return nil with AbsNone}}.
Proof.
    (* Break up the while portion of the loop *)
    unfold loop. unfold afterWhile. unfold afterInitAssigns.
    (* WHILE ALnot (!T === A0) DO *)
    eapply strengthenPost.
    eapply whileThm with (invariant := loopInv). unfold loopInv.
    eapply strengthenPost.
    (* N ::= !P; *) eapply compose. pcrunch.
    (* NEW P,ANum(Size_l);*) eapply compose. pcrunch. simp. simp. simp.
   (* CStore ((!P)+++(ANum F_p)) (!T)) *) eapply compose. pcrunch. apply treeRef1.
                                           apply H. apply H0.
   (* CStore ((!P)+++(ANum F_n)) (!N)) *) eapply compose. pcrunch. apply treeRef2.
                                           apply H. apply H0. simp.
    (* CLoad Tmp_l ((!T)+++ANum(F_l)) *) eapply compose. pcrunch.
    (* CLoad Tmp_r ((!T) +++ A1) *) eapply compose. pcrunch.
    (* 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. pcrunch. eapply deleteExists1. apply H0.
            (* I ::= !Tmp_l *) pcrunch. pcrunch. pcrunch. pcrunch. pcrunch.
        (* FI *)
        apply mergeTheorem1.
    (* ELSE *)
        (* (CIf (!Tmp_l === A0) *) simpl. eapply if_statement.
            (* CLoad T (!T +++ A1) *) simpl. pcrunch.
        (* ELSE *)
            (* CIf (!Tmp_r === A0) *) simpl. eapply if_statement.
                (* CLoad T (!T +++ A0) *) simpl. pcrunch.
            (* ELSE *)
                (* N ::= !I *) simpl. eapply compose. pcrunch.
                (* NEW I, A2 *) eapply compose. pcrunch.
                (* CStore (I ++++ A0) (!N) *) eapply compose. pcrunch.
                                          apply storeCheck1. apply H. apply H0.
                (* CLoad Tmp_l (! T +++ A1) *) eapply compose. pcrunch.
                (* CStore (! I +++ A1) (! Tmp_l) *) eapply compose. pcrunch.
                                          apply storeCheck2. apply H. apply H0.
                (* (CLoad T (! T +++ A0) *) pcrunch.
            (* FI *) pcrunch. pcrunch. pcrunch. pcrunch. pcrunch. pcrunch.
                  apply mergeTheorem2.
        (* FI *) pcrunch. apply mergeTheorem3.
    (* FI *) pcrunch. apply mergeTheorem4.
    pcrunch. pcrunch. pcrunch. pcrunch. pcrunch. pcrunch.
    apply implication1. intros. inversion H. intros. inversion H.
    apply implication2. apply implication3. intros. apply H. intros. inversion H.
Qed.
```

Figure 2: Top level Coq proof of our program.

A foundation for structured proof development The top level proof of a program verification always models the structure of the program, and can mostly be generated automatically. The only places where the user would need to fill in information is for loop invariants and for the resulting state after a merge at the end of an if-then-else block. Most of the complicated theorem proving work is moved into lemmas. When the top level proof is done, a precise statement of the abstract state is defined for each line of code.

To facilitate automation, we introduce several features. For example, `absUpdateVar` *s i val* and `absUpdateLoc` *s l val* in our separation logic indicate that a variable or location has changed. This avoids doing substitutions which may require unfolding or other complex tasks. The task of simplifying out these constructs is pushed into lemmas at the end of a loop or if-then-else statement block. Structured proof development lays the ground work for better automation as well as for lightweight verification in which some of the generated lemmas are `admit`-ted.

2 A SAMPLE VERIFICATION

PEDANTIC excels at reasoning about cross-structure invariants, and we show this with a small PEDANTIC verification example.

Figure 1 shows a C-like program which performs a traversal of a tree, given in the parameter `r` to the function `build_pre_order` and places the result in a linked list `p`. This function contains a pointer `t` which walks the tree. As the tree is walked, elements are added to the `p` list.

Our Coq proof verifies a number of important properties of the data structures in this program, including: (1) the program maintains two well formed linked lists, the heads of which are pointed to by `n` and `p`; (2) the program maintains a well formed tree pointed to by `r`; (3) `t` always points to an element in the tree rooted at `r`; (4) the two lists and the tree are disjoint in memory; (5) no other heap memory is allocated; and, (6) the `t` field of every element in both list structures points to an element in the tree. Invariant 6, in particular is a cross-structure invariant which PEDANTIC is highly suited to verify.

2.1 The state assertion

To better understand state assertions we present a sample heap snapshot of our example program in Figure 3. This example shows the heap memory after two loop iterations. To the right of the trees we give the general state invariant for the loop as we formalize it in Coq. The three `TREE` assertions characterize the tree and the two lists of the heap. The four parameters of `TREE` are: (1) the root of the tree; (2) a variable holding an equivalent functional representation (discussed below); (3) the word size of a record; (4) and, a list of offsets for all the child node pointers. Here the tree needs two such offsets, $[0, 1]$ and the lists only need one, $[0]$.

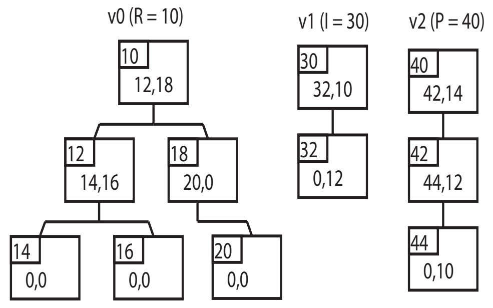
The functional representations $v_0/v_1/v_2$ defined by `TREE` characterize both the recursive list/tree structure and additionally embed information on the pointer structure, similar to methods used in [20]. For the snapshot from the Figure, v_0 would be

$[10, ([12, [14, [0], [0]], [16, [0], [0]]]), ([18, [20, [0], [0]], [0]])]$

Getting back to the invariant assertion, following the `TREE` assertions, the two `AbsAll` clauses assert for each list that the pointers in the second field of each node in the list point into the tree. `TreeRecords` is a function that returns the list of record pointers given a functional representation. For example, for the representation above, `TreeRecords` returns $[10, 12, 14, 16, 18, 20]$. x `inTree` y is shorthand for $x \in$ `TreeRecords`(y), and `find` takes an address in a tree and a functional representation of the tree (as shown above), and returns the subtree rooted at that address. For example, `find`$([30, [32, [0], 12], 10], 32)$ will return $[32, [0], 12]$. Finally, the last line states that either `T` is `0` or that `T` points to an element in `R`.

2.2 Verifying the assertions

Figure 2 shows the top level Coq proof that the invariant holds throughout the program in figure 1. Notice that the proof follows the structure of the program. This main proof generates 12 lemmas that need to be verified. `treeRef1` and `treeRef2` verify that a heap location being read belongs to an allocated block. `storeCheck1` and `storeCheck2` similarly verify that a heap location being written to is in an allocated block. `deleteExists1` verifies that a block being deallocated is actually an allocated block on the heap. `mergeTheorem1`, `mergeTheorem2`, `mergeTheorem3` and `mergeTheorem4` are used to merge the resulting states from



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

Figure 3: A snapshot of a possible heap state of the program from Figure 1, and the general PEDANTIC assertion describing the state invariant for the loop over all executions.

forward propagating the then and else branches of an if-then-else. The theorems `implication1`, `implication2`, `implication3` verify that the state generated by forward propagation matches the post-condition at the end of a block.

2.3 Coq source files

The main proof described in this paper with its 12 lemmas can be found in `TreeTraversal.v`. Note that translation form C to the Coq imperative language was done by hand. There is no C parser front end. We also included the file `SatSolverMain.v` which shows the incomplete DPLL solver verification. It also includes `SatSolver.c` and a DIMACS file `test` which gives the original C implementation of our Sat solver (before we hand translated it to our Coq syntax). The files for both CoqPIE and PEDANTIC are found at `https://github.com/kendroe/CoqPIE`. They compile with Coq 8.5pl3.

3 RELATED WORK

Separation logic was originally developed by John Reynolds [30]. Tools based on separation logic were developed and found to be effective in finding bugs in Microsoft device drivers [13, 15, 27]. McCreight developed a set of Coq axioms and tactics to reason about separation logic [25]. Systems such as Charge! [4], Bedrock [1] and VST [2, 3] were then developed to provide Coq frameworks for verifying imperative programs with recursive data structures. Coq's Ltac tactic language is limited and restricts the ability automate proofs in these systems; PEDANTIC uses a deep model to give greater flexibility in designing tactics. VST was recently used to verify the correctness of the HMAC algorithm in OpenSSL [5]. This is an approximately 200-line C program. This verification task was quite tedious and emphasizes the need to study proof development productivity issues. VST is built on top of Compcert-C [17] and hence the C language semantics in VST are fully verified.

Also, recently, a system called IRIS [31] which uses concurrent separation logic has been developed. Being new, so far, they only have written up small examples.

4 CONCLUSION

This paper covered the basic concepts of PEDANTIC, a tool for verifying the correctness of C programs. We have demonstrated how dynamic data structure invariants including cross referenced dependencies can be expressed and reasoned about. There are many other features for which there is not enough space to discuss. We currently have an implementation of the framework that contains all of the basic data types and tactics but without correctness proofs of the tactics. The proof of the example program invariant of Figure 3 is complete. We are currently working on the verification of the DPLL [11] based SAT solver invariant. While not complete, it is well under way and many of the issues of scaling PEDANTIC for more complex verification tasks such as the performance of Coq on large theorems have already been addressed. The DPLL verification also demonstrates that automatically finding invariants is infeasible. The invariants are complex and represent non-obvious properties of the program.

Our larger goal is greater automation of the proof development process. Much of the work done for the tree traversal proof presented can be automated by a tool such as CoqPIE [32]. The top level `loopInvariant` theorem can be generated automatically for the most part. The two areas that need user input are 1) providing loop invariants and 2) at merge points, some specification of the output state is required. Of the 12 lemmas, many turn out to be straightforward and can be automated.

The biggest challenge in making interactive theorem proving tools practical for software development is to better understand proof development productivity issues. Right now, based on the author's experience using Coq, every hour of software development likely requires 100 hours of proof development time. This ratio needs to come down. However, once these issues are addressed, proof development will become an important part of software development methodologies as it can find bugs for which there is otherwise no effective methodology.

REFERENCES

[1] Greg Morrisett Avraham Shinnar Ryan Wisnesky Adam Chlipala, Gregory Malecha. Effective interactive proofs for higher-order imperative programs. 2009.

[2] A. W. Appel. Tactics for separation logic, 2006. Early draft.

[3] Andrew Appel and Sandrine Blazy. Separation logic for small-step cminor. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. 2007.

[4] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - a framework for higher-order separation logic in Coq. In *Third International Conference, ITP*, 2012.

[5] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. Verified correctness and security of openssl hmac. In *USENIX Security*, volume 15, pages 207–221, 2015.

[6] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[7] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *PLDI*, pages 431–447, 2016.

[8] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd Programming Language Design and Implementation (PLDI)*, 2011.

[9] David Costanzo, Zhong Shao, and Ronghui. End-to-end verification of information-flow security for c and assembly programs.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[11] D. Ranjan D. Tang, Y. Yu and S. Malik. Analysis of search based algorithms for satisfiability of quantified boolean formulas arising from circuit state space diameter problems. *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, May, 2004.

[12] Jyotirmoy V. Deshmukh, E. Allen Emerson, and Prateek Gupta. Automatic verification of parameterized data structures. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 27–41, 2006.

[13] Peter W. O'Hearn Dino Distefano and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[14] S. P. Rahul Westley Weimer George C. Necula, Scott McPeak. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, March 2002.

[15] B Cook D Distefano PW O'Hearn T Wies J Berdine, C Calcagno and H Yang. Shape analysis for composite data structures. In *CAV'07*. Springer, 2007.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[17] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[18] Huisong Li, Franois Brenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. page 13 pages, 2017.

[19] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.

[20] Gregory Malecha and Greg Morrisett. Mechanized verification with sharing. In *ICTAC*, 2010.

[21] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM.

[22] Roman Manevich, Boris Dogadov2, and Noam Rinetzky. From shape analysis to termination analysis in linear time. In *CAV*, 2016.

[23] Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Information and Media Technologies*, 4(2):304–316, 2009.

[24] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. 2006.

[25] Andrew Mccreight. Practical tactics for separation logic. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 343–358, 2009.

[26] ThanhVu Nguyen. *Automating Program Verification and Repair Using Invariant Analysis and Test-input Generation*. PhD thesis, University of New Mexico, August 2014.

[27] Hongseok Yang Peter O'Hearn, John Reynolds. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume LNCS 2142, pages 1–19, 2001.

[28] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. 2011.

[29] Piskac R., Wies T., and Zufferey D. Automating separation logic using smt. In Sharygina N. and Veith H., editors, *Computer Aided Verification. CAV. Lecture Notes in Computer Science*, volume 8044. Springer, Berlin, Heidelberg, 2013.

[30] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[31] Ales Bizjak Jacques-Henri Jourdan Derek Dreyer Robbert Krebbers, Ralf Jung and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723, 2017.

[32] Kenneth Roe and Scott Smith. Coqpie: An ide aimed at improving proof development productivity (rough diamond. In *ITP*, 2016.

[33] Tahina Ramananandro Zhong Shao Xiongnan (Newman) Wu Shu-Chun Weng Haozhong Zhang Yu Guo Ronghui Gu, Jrmie Koenig. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608, 2015.

[34] Reinhard Wilhel Shmuel Sagiv, Thomas W. Reps. Solving shape-analysis problems in languages with destructive updating. 20(1):1–50, 1998.

[35] Yavuz-Kahveci T. and Bultan T. Automated verification of concurrent linked lists with counters. In Hermenegildo M.V. and Puebla G., editors, *Static Analysis. SAS. Lecture Notes in Computer Science*, volume 2477, 2002.

[36] Ting Zhang Zohar Manna, Henny B. Sipma. Verifying balanced trees. In *Proceedings of the Symposium on Logical Foundations of Computer Science (LFCS 2007)*, 2007.