Refinements to techniques for verifying shape analysis invariants in Coq

Kenneth Roe and Scott Smith
The Johns Hopkins University

Abstract. We describe the PEDANTIC framework for verifying the correctness of C-like programs using Coq. PEDANTIC is designed to prove invariants over complex dynamic data structures such as inter-referencing trees and linked lists. The PEDANTIC tactic library has been constructed to allow program verifications to be done with reasonably compact proofs. We introduce a couple of important innovations. First, we introduce constructs to allow for elegant reasoning about cross referencing relationships between recursive data structures on the heap. Second we have designed our framework to be extensible in spite of its use of a deep embedding of the logic into Coq. Our data structure for representing state assertions is parameterized by functions which give semantic interpretations to the constructs. This allows us to extend our system with new predicates and functions without changing the underlying Coq data structures that define the core PEDANTIC logic.

1 Introduction

This paper presents first steps towards a framework for verifying programs in imparative languages such as C that use complex heap based data structures. Our extensible PEDANTIC (Proof Engine for Deductive Automation using Nondeterministic Traversal of Instruction Code) framework is implemented in Coq and is based on ideas from many other separation logic frameworks such as [2, 1, 3]. 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.

While we use Coq as our underlying prover as do the aforecited works, we make some different choices in terms of how the logic is structured. 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 at the end of an if-then-else construct; and (4) use of a `crunch` tactic to perform low level theorem proving tasks.

Deep model Unlike the frameworks of [2, 1, 3], PEDANTIC uses a deep embedding. This means an explicit Coq data structure is defined for the assertion language grammar, whereas the cited frameworks use higher-order abstract syntax (HOAS). The trade-offs between the two approaches are well-known, but we briefly review them in the context of our goals. We use a deep embedding because Coq's Galina language functions can then be used to manipulate the assertions and define some of the tactics. While Ltac can be used in the shallow model to create arbitrary tactics, Ltac requires a proof to be constructed even

if there is a simpler algorithm of computing the outcome of the tactic. With the deep model, one can do proofs of the Galina functions themselves which eliminates the need to do this work in the program verification. There is the disadvantage that a deep model is less flexible. In the above systems, one can add complex specifications by simply defining new functions and properties as part of the proof. To address this shortcoming, we have developed a novel approach in which we parameterized the data structures for representing the state, allowing users to add new definitions by redefining semantic functions. While this is more complex than what needs to be done with a shallow model, we believe a preprocessor could be written to automate the additional steps. The deep model also means some functionality embedded in Coq needs to be reimplemented. For example, Some arithmetic and logical simplifications need to be re-implemented whereas in a shallow embedding they come for free since Coq's logical reasoning tactics can automatically apply. There is no getting around this weakness but we believe the reward will be worth the price. Note that once the deep embedding syntax is unfolded to its underlying meaning as defined in Coq, the difference between the two approaches has vanished.

Embedding pointers in a functional representation The Btree example in [3] 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. We also designed our logic to supports the expression of data structure invariants in semi-independent layers. First, a shape invariant is used to describe the recursive data structure. Then, separate predicates can be used to describe additional constraints, giving state assertions more modularity.

Merge 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` tactics to align recursive data structures to facilitate the merge. We also use this pairing technique for proving assertion entailment properties.

Crunch Finally, we found it useful to create a series of `crunch` tactics to automate many simple operations. For example, if one sees a hypothesis of the form `H: None = Some _`, the tactic `inversion H` can be immediately applied to solve the goal. We have rolled up a set of these rules into a single large Ltac match statement.

1.1 A program example

PEDANTIC excels at reasoning about cross-structure invariants, and we give a small program example here which contains such invariants that PEDANTIC can verify. 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

```
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 }
```

Fig. 1. Example program which builds a linked list of all nodes in a tree

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 Separation Invariants

In this section we describe the internal structure of PEDANTIC in more detail.

Shallow embeddings such as Bedrock[2] allow one to create arbitrary recursive data structure invariants by defining a Coq function that can be used in a separation logic based assertion. With a deep embedding where the logic is defined as a data structure (rather than a Coq expression), it is non-trivial to make the framework extensible. To address this problem, we make PEDANTIC an explicitly parametric logic consisting of a generic core logic which can be specialized in several dimensions for particular program structures.

So, in PEDANTIC we define a general *abstract state* logic data structure, which can be instantiated to a particular *state instance* depending on the particular kinds of inductive program invariants that are needed. The abstract syntax for states is shown in Figure 2. `absExp` is the type of generic assertion expressions, and `absState` is the type of generic state assertions. These types

```
Inductive absExp {ev} {eq : ev -> ev -> bool}
 { f : id -> list (@Value ev) -> (@Value ev) } : Type :=
 | AbsConstVal : (@Value ev) -> absExp
 | AbsVar : id -> absExp
 | AbsQVar : absVar -> absExp
 | AbsFun : id -> list absExp -> absExp.

Inductive absState {ev} {eq : ev -> ev -> bool}
 { f : id -> list (@Value ev) -> (@Value ev) }
 { t : id -> list (@Value ev) -> heap -> Prop }
 ... :=
 | AbsExists : (@absExp ev eq f) -> @absState ev eq f t ac -> absState
 | AbsAll : (@absExp ev eq f) -> @absState ev eq f t ac -> absState
 | AbsCompose : @absState ev eq f t ac -> @absState ev eq f t ac -> absState
 | AbsEmpty : absState
 | AbsLeaf : id -> (list (@absExp ev eq f)) -> absState
 ...
```

Fig. 2. Separation expression and assertion data structures, from file `AbsState.v`

$$
\begin{array}{ll}
(e,h) \vdash [P] & \text{iff } \ll P \gg e \neq 0 \text{ and } dom(h) = \emptyset \\
(e,h) \vdash l \mapsto v & \text{iff } h(\ll l \gg e) = \ll v \gg e \wedge \forall x \in \mathcal{N}, x \neq \ll l \gg e \rightarrow x \notin dom(h) \\
(e,h) \vdash \texttt{TREE}(r,v,n,\bar{f}) & \text{iff } (\ll r \gg e = 0 \wedge h = \emptyset) \vee \\
\quad \exists h_0,\ldots,h_{n+m},x_0,\ldots,x_{n-1},v_0,\ldots,v_m.\ll r \gg e \neq 0 \wedge & \\
\quad (e,h_0) \vdash (\ll r \gg e+0) \mapsto x_0 \ \wedge \ldots \wedge (e,h_{n-1}) \vdash (\ll r \gg e+n-1) \mapsto x_{n-1} \wedge & \\
\quad (e,h_n) \vdash \texttt{TREE}(h(\ll r \gg +f_0), v_{f_0}, s, \bar{f}) \ \wedge \ldots \wedge & \\
\quad (e,h_{n+m-1}) \vdash \texttt{TREE}(h(\ll r \gg +f_{m-1}), v_{f_{m-1}}, s, \bar{f}) \wedge & \\
\quad \texttt{combine}(h,h_0,\ldots,h_{n+m-1}) \wedge & \\
\quad v = [\ \ll r \gg e, v_0,\ldots,v_{n-1}] \wedge \forall i < n.i \notin \bar{f} \text{ implies } v_i = x_i &
\end{array}
$$

where $\texttt{combine}(h,h_0,\ldots,h_n)$ iff $\forall i,j \leq n.dom(h_i) \cap dom(h_j) = \emptyset \ \wedge$
$\forall i \in dom(h).\exists j.i \in dom(h_j) \wedge h(i) = h_j(i)$

Fig. 3. Semantics for the separation predicates of the `basicState` instantiation of `absState`. (e,h) represents a concrete state. $(e,h) \vdash s$ means that (e,h) satisfies the assertion s. $e \in \texttt{Id} \rightarrow \texttt{Nat}$ represents assignments for program variables and $h = \texttt{Nat} \rightarrow \texttt{optionNat}$ represents the values on the heap. The notation $\ll exp \gg e$ represents evaluation of the expression *exp* with the set of variables in the environment e.

are parametric in function parameters `f`/`t`/... which are provided to define an *instance* of the abstract semantics. The `f` parameter in `absExp` is used by the `absFun` clause to define particular arithmetic operators as needed. The `t` parameter in `AbsState` is used by the `AbsLeaf` clause, and allows arbitrary new state assertions to be plugged into the abstract state assertion grammar. We use the abbreviation $a * b$ for `AbsCompose` a b. We use de Brujin indices for bound variables, so `AbsAll`/`AbsExists` do not specify the names of the variable they introduce. We use sugared quantifier notation with variables $v_0, v_1, \ldots$ below for readability.

For the particular example of this paper, the `absExp` and `absState` datatypes are instantiated to the corresponding `basicEval` and `basicState` datatypes by providing explicit `f` and `t` functions. We instantiate with an `f` that defines standard operations on integers; parameter `t` is instantiated to provide

three important leaf predicates: `AbsPredicate`, `AbsCell` and `AbsTree`. `AbsLeaf AbsPredicate` P, denoted $[P]$, indicates an arbitrary predicate on the heap, `AbsLeaf AbsCell` l v, abbreviated $l \mapsto v$, asserts that cell l has contents v, and `AbsLeaf AbsTree`, abbreviated `TREE`, is used for characterizing recursive data structures on the heap, either lists or trees. By defining a fixed grammar for tree structure assertions we can in tandem define generalized fold and unfold tactics which will work over different recursive data structures in different derivations that are defined with `AbsTree`. The semantics for these three `t` extensions are given in Figure 3; the semantics of the `absState` core seperation logic is standard and is elided.

The `basicState` instance is itself further extensible, a user could create a `userState` and `userEval` to integrate additional functions. These functions must support all the functionality of `basicState` and `basicEval`; to verify this holds, there is a `supportsFunctionality` definition that needs to be instantiated and proven.

2.1 An example state assertion

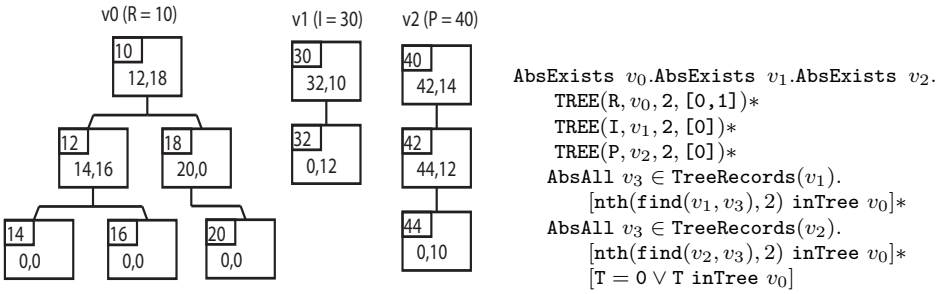

Fig. 4. A snapshot of a possible heap state of the program from Figure 1, and the general PEDANTIC assertion describing the state invariant for the loop over all executions.

To better understand state assertions we present a sample heap snapshot of our example program in Figure 4. This example shows the heap memory after two loop iterations have been completed. 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 [3]. For the snapshot from the

Figure, v_0 would for example be the list

```
[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 pointer to a tree, 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 `T`.

2.2 The Coq Proofs

The Coq files associated with this paper include file `TreeTraversal.v` which contains a commented set of steps of the proof that describe the underlying approach to verification in PEDANTIC.

3 Conclusion

This paper has covered the basic ideas in the PEDANTIC framework, 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 in the paper to discuss. We currently have an implementation of the framework that contains all of the basic data types and tactics. The proof of the example program invariant of Figure 4 is complete, but correctness proofs of most auxiliary Lemmas still need to be completed and are for now `admit`ted. Once verification of the Lemmas is complete, we plan to apply PEDANTIC to a DPLL based SAT solver.

Acknowledgements The authors would like to thank Greg Malecha for his feedback on this paper.

References

1. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - a framework for higher-order separation logic in Coq. In *Third International Conference, ITP*, 2012.
2. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd Programming Language Design and Implementation (PLDI)*, 2011.
3. Gregory Malecha and Greg Morrisett. Mechanized verification with sharing. In *ICTAC*, 2010.
4. Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Information and Media Technologies*, 4(2):304–316, 2009.