Refinements to techniques for verifying shape analysis invariants in Coq

Kenneth Roe

The Johns Hopkins University

Abstract. The research in this proposal is aimed at creating a theorem proving framework that will be practical for many large software systems. The system is aimed at finding bugs that corrupt data structures. Program data structures can be quite complex and involve many invariants relating the different pieces.
Our verification framework is called PEDANTIC. It is designed for verifying the correctness of C-like programs and is built on top of the Coq theorem prover. 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. Deep embedding means creating custom data structures to represent the assertions rather than directly encoding them in the Coq theorem proving language. This gives greater flexibility in designing tactics. 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

One of the key challenges in any large scale software development project is that of finding bugs. Any production quality piece of software goes through an extensive testing process before being shipped to customers. Yet this process is often insufficient to catch all bugs. Many critical bugs only show up after a customer starts using a product. Often these bugs are intermittent. While these bugs can be traced to defects in the code, many variables in the environment cause the software to execute differently each run making these bugs hard to track. A company can spend several weeks trying to find a reliable way to reproduce the bug before finding a fix.

Many security flaws can also be traced to software bugs. Stack smashing is a common technique for a malicious piece of code to take control of a computer. In order for stack smashing to work, there must be a missing overflow check in the software. Stack smashing involves allowing a buffer to overflow and corrupt the return address placed on the stack from when a function was called. This bogus return address causes malicious code entered by the attacker to be executed.

One interesting property of many of these bugs is that they somehow corrupt a program's data structures. Large programs tend to have complex data structures with the same information being represented in many different forms to optimize either space usage or speed. The goal of the research is to extend formal verification techniques to handle these complex data structure relationships. The idea is to construct a set of mathematical statements describing the invariants. Then, using formal reasoning techniques, step through the code of a program to verify that it maintains the integrity of the data structures.

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. In our design, data structure invariants need to be entered by the user. Our experience shows that the size of these invariants tend not to be larger than the size of the program being verified. One key issue in many formal systems is that of inferring loop invariants. Our experience has been that loop invariants are usually very similar to the pre- and post- condition invariants. Since the task of automatically generating a loop invariant can be fairly difficult, we have made the decision to require the user to enter all loop invariants. Even at this point, the task of performing forward inferences in programs cannot be fully automated though many simple can be. Often data structures need to be unfolded if an instruction references a field inside. In some cases data structure unfolding can be avoided by some specialized propagation rules. Making the choice about doing the unfolding or using the specialized rule needs to be done by the user. Another area where the user has to assist the theorem prover as at the end of a sequence of statements, there will be a post condition (which often looks very much like the precondition). However, the mechanical steps performed in the forward chaining produce an assertion that does not look like the post-condition. The user will likely have to perform many manual theorem proving steps to prove that the result of the forward propagation implies the post condition.

We are using the Coq theorem prover to implement our system. Coq is a strongly typed higher order interactive theorem prover. One of our primary reasons for choosing Coq is that it can reason about recursive functions and data structures.[2, 3, 14]

1.1 A program example

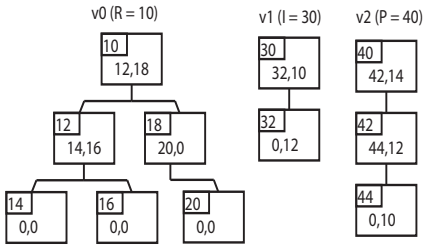
PEDANTIC excels at reasoning about cross-structure invariants, and we give a small program example here which contains such invariants that PEDANTIC can verify. Cross structure invariants are invariants that related the information in different data structures. Figure 1 shows a C-like program which performs 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

```
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

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.



$$
\begin{aligned}
&\exists v_0.\exists v_1.\exists v_2.\\
&\quad \texttt{TREE}(\texttt{R}, v_0, 2, \texttt{[0,1]})*\\
&\quad \texttt{TREE}(\texttt{I}, v_1, 2, \texttt{[0]})*\\
&\quad \texttt{TREE}(\texttt{P}, v_2, 2, \texttt{[0]})*\\
&\quad \forall v_3 \in \texttt{TreeRecords}(v_1).\\
&\qquad [\texttt{nth}(\texttt{find}(v_1, v_3), 2)\ \texttt{inTree}\ v_0]*\\
&\quad \forall v_3 \in \texttt{TreeRecords}(v_2).\\
&\qquad [\texttt{nth}(\texttt{find}(v_2, v_3), 2)\ \texttt{inTree}\ v_0]*\\
&\quad [\texttt{T} = 0 \vee \texttt{T}\ \texttt{inTree}\ v_0]
\end{aligned}
$$

Fig. 2. 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.

Invariant 6, in particular is a cross-structure invariant which PEDANTIC is highly suited to verify.

2 The assertion language

We present our assertion language through the use of an example. Figure 2 shows the invariant for our program as well as a picture of a sample state. 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.

h_0	$\{10 \mapsto 12, 11 \mapsto 18, 12 \mapsto 14, 13 \mapsto 16, 14 \mapsto 0, 15 \mapsto 0, 16 \mapsto 0, 17 \mapsto 0, 18 \mapsto 20,$ $19 \mapsto 0, 20 \mapsto 0, 21 \mapsto 0\}$
h_1	$\{30 \mapsto 32, 31 \mapsto 10, 32 \mapsto 0, 33 \mapsto 12\}$
h_2	$\{40 \mapsto 42, 41 \mapsto 14, 42 \mapsto 44, 43 \mapsto 12, 44 \mapsto 0, 45 \mapsto 10\}$
$h_1 \cup h_2 \cup h_3$	$\{10 \mapsto 12, 11 \mapsto 18, 12 \mapsto 14, 13 \mapsto 16, 14 \mapsto 0, 15 \mapsto 0, 16 \mapsto 0, 17 \mapsto 0, 18 \mapsto 20,$ $19 \mapsto 0, 20 \mapsto 0, 21 \mapsto 0, 30 \mapsto 32, 31 \mapsto 10, 32 \mapsto 0, 33 \mapsto 12, 40 \mapsto 42, 41 \mapsto 14,$ $42 \mapsto 44, 43 \mapsto 12, 44 \mapsto 0, 45 \mapsto 10\}$
e	$\{R \mapsto 10, I \mapsto 30, P \mapsto 40\}$
v_0	`[10,([12,[14,[0],[0]],[16,[0],[0]]]),([18,[20,[0],[0]],[0]])]`
v_1	`[30,([32,[0],12]),10]`
v_2	`[40,([42,[44,[0],10],12]),14]`

Fig. 3. This figure shows values for the variables of the assertion on the right in figure 2 that correspond to the snapshot at the left. It shows how the separation * works. Here $(e, h_0) \models$ `TREE`$(\texttt{R}, v_0, 2, \texttt{[0,1]})$, $(e, h_1) \models$ `TREE`$(\texttt{I}, v_1, 2, \texttt{[0]})$, $(e, h_2) \models$ `TREE`$(\texttt{P}, v_2, 2, \texttt{[0]})$ and $(e, h_1 \cup h_2 \cup h_3) \models$ `TREE`$(\texttt{R}, v_0, 2, \texttt{[0,1]}) * $ `TREE`$(\texttt{I}, v_1, 2, \texttt{[0]}) * $ `TREE`$(\texttt{P}, v_2, 2, \texttt{[0]})$.

The model for our assertion logic Before discussing the state assertion itself, we need to discuss our underlying model for representing program state. A program state consists of two parts, a variable store (the environment) and a heap. The variable store is modeled as a function mapping variable names to values $v \to \mathcal{N}$. Values in our model are always non-negative natural numbers. This does not exactly match what is in C but works well for the types of theorems we are trying to prove. The heap is modeled by another function $\{n : \mathcal{N} | n > 0\} \to \mathcal{N} \cup \perp$. Locations are modeled by a mapping of non-zero positive numbers to either a natural number (the value in the heap location or $\perp$ if the location is not allocated). For now, the model does not deal with function calls. This will require some extensions. We often refer to this model using the tuple (e, h).

The assertion language Assertions are logical predicates over our model. The predicates use separation logic (which is an extension of first order logic). Key to separation logic is the * operator which combines disjoint pieces of an assertion. It is formally defined as follows:

Definition 1. *We say* $(e, h) \models s_1 * s_2$ *if and only if there exists* h', h'' *such that* $(e, h') \models s_1$ *and* $(e, h'') \models s_2$ *and* $dom(h_1) \cap dom(h_2) = \emptyset$ *and* $h = h' \cup h''$.

The idea with the * operator is that it can put together predicates describing smaller disjoint heaps into larger heaps. The three `TREE` assertions on the right in figure 2 characterize the tree and the two lists pictured at the list. They are put together with *. A `TREE` predicate on its own is satisfied by a model with a smaller heap containing just its own list or heap. The * predicate puts them together to represent a heap with all three trees. Figure 3 shows a how * can put together two heaps.

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]`. Figure 3 shows values for the parameters of `TREE` that are needed for it to hold.

In order for the `TREE` predicate to be true, in addition to other criteria, the values of the variables $v_0/v_1/v_2$ must be the alternate functional representation to for the trees stored in the corresponding heaps, $h_0/h_1/h_2$ in figure 3. If we look at the figure, h_1 is the heap that corresponds to a linked list that contains the values 10 and 12 for which the first node is at location 30 and the second at location 32. The representation of the list in this format is difficult to read. It turns out that it is also hard for the theorem prover to reason about the list in this form. We have created a second representation of the information as shown by v_1 in figure 3. It contains both the values of the list (10 and 12) as well as the memory locations of each of the records of the list are stored (30 and 32). It is represented by a nested list data structure. This functional representation facilitates the use of structural induction (or recursion).

Getting back to the invariant assertion, following the `TREE` assertions, the two $\forall$ 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`.

3 Theorem proving

Now that the assertion language has been discussed, we now introduce our theorem proving framework for verifying the assertions. Figure 4 shows part of the proof tree that from the entry pre-condition, the first few statements of the program set up the main invariant. Most goals in our system take the form of a Hoare triple, $\{pre\}$`code`$\{post\}$. This triple is an assertion that if *pre* is true before executing `code`, that *post* will be true afterwards. The figure shows a few of the steps taken. The top level goal is broken into two subgoals. One should note the introduction of an *existential* variable ?1234. Coq allows for the use of existential variables in the construction of a proof tree. The idea is that ?1234 will be replaced with a concrete term later on in the proof process. In this case, ?1234 will be instantiated to $\exists v_0.$`TREE`$(\texttt{R}, v_0, 2, \texttt{[0,1]}) * [T = R] * [I = 0] * [P = 0]$ in order to make the proof of the goal at the bottom left in figure 4 trivial.

3.1 Goals

Most of the proof goals in our system are of the form of Hoare triples. There are some goals (such as those at the two leaves in figure 4) which are implication goals. Finally, merge goals appear frequently. When propagating over a statement

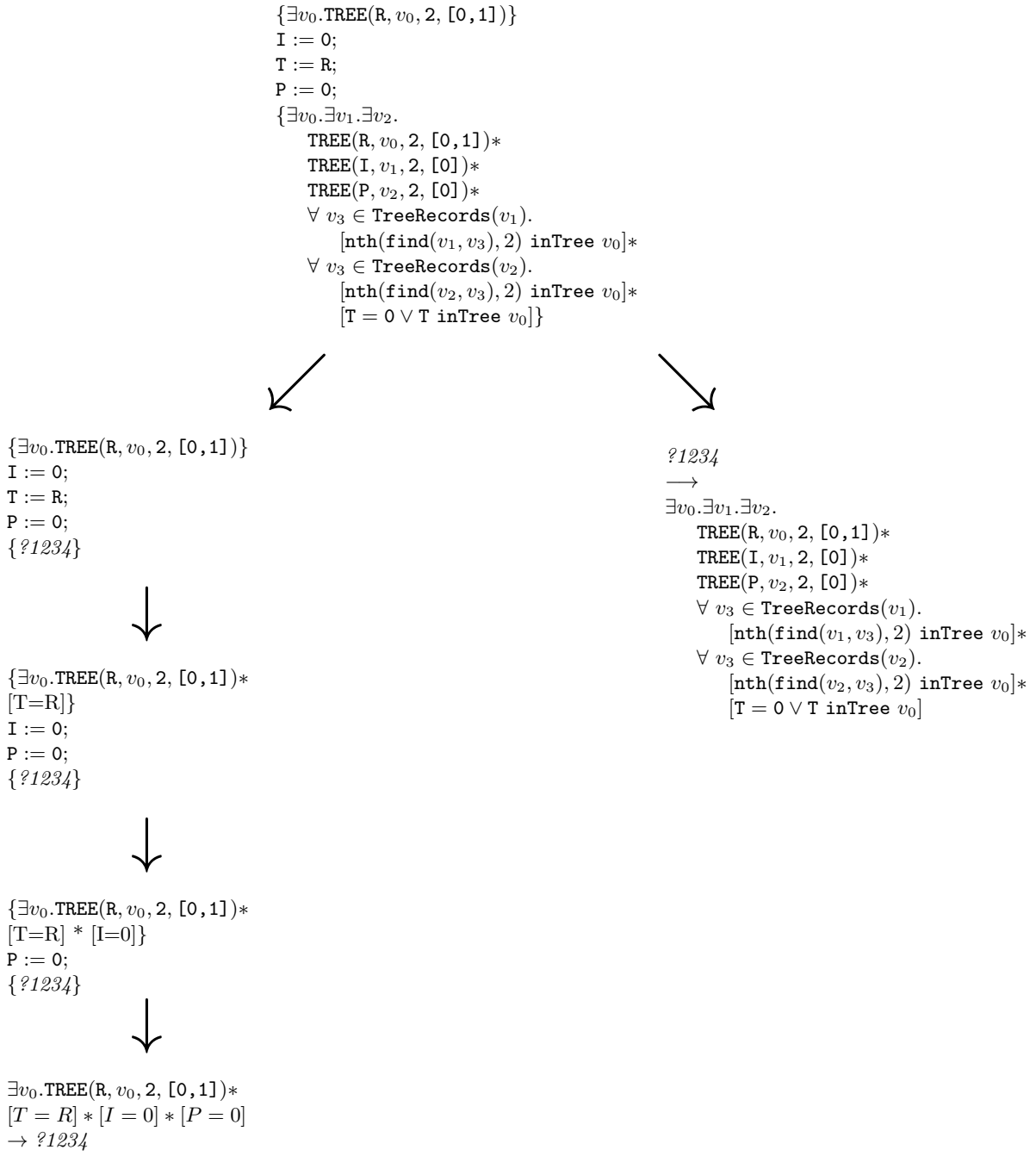


Fig. 4. Partial proof tree for the initialization statements

of the form `if` c `then` s_1 `else` s_2, one will create two assertions $post_1$ and $post_2$ corresponding to propagations over s_1 and s_2. There will be a goal at the end of the form $post_1 \rightarrow ?2345 \wedge post_2 \rightarrow ?2345$. Solving this goal involves coming up with a meaningful assertion for ?2345. We have an algorithm that creates this term by first pairing off the identical parts of $post_1$ and $post_2$ and then using more specialized rules to combine the pieces that remain.

3.2 Proof tactics

We saw in figure 4 use of the forward propagation rule to walk through the statements of a program. In addition, our framework contains many other types of tactics. There are specialized tactics for solving the merge goals described above, there are tactics for solving implication goals such as what ends up in the bottom right after ?1234 is instantiated. These implication tactics (like the merge tactics) work by first pairing off identical parts of the premise and consequent states and then use more specialized rules for the remaining pieces.

There are tactics for simplifying states and finally there are tactics for folding and unfolding. Consider the goal at the top in figure 5. Before we can forward propagate, we first must do an unfolding. We should note that after unfolding (or the application of many other tactics), it is useful to simplify the result. for example, the sub term $nth([I, v_0, v_1], 1)$ that appears in the middle goal of figure 5 can be simplified to v_0.

4 Deep model

Our framework uses a deep embedding. This means an explicit Coq data structure is defined for the assertion language (rather than defining the language by creating function definitions built on top of the Coq data structures) 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 (Coq's meta-language for defining tactics) 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. We illustrate the difference by showing a deep model for arithmetic involving addition and multiplication. In a shallow model, proving a theorem such as $a+c = a+b+c-b$, would be entered directly in Coq and proven. One would have to find tactics in Coq to reorganize the order of the terms on the right and cancel out the $+b$ and $-b$ terms. In a deep model, one would first define a data structure for representing arithmetic expressions:

`type` $expr$ = `Const` int | `Var` id | `Plus` $expr \times expr$ | `Minus` $expr \times expr$ | `Times` $expr \times expr$

Next, one would write an evaluation function to give the model semantics. This function might look something like this:

$\exists v_0 \exists v_1 \exists v_2 [Tmp\_l = 0] * [l \neq 0] * [tmp\_r = 0] *$
$[Tmp\_r = 0 \vee Tmp\_r \in \texttt{TreeRecords}(v_0)] *$
$[nth(nth(find(v_0, T)), 2), 0) = (Tmp\_r)] *$
$[nth(nth(find(v_0, T)), 1), 0) = 0] *$
$[T \in \texttt{TreeRecords}(v_0)] *$
$P + 0 \mapsto N * P + 1 \mapsto T * [T \neq 0] *$
$\texttt{TREE}(\texttt{R}, v_0, 2, \texttt{[0,1]}) * \boxed{\texttt{TREE}(\texttt{I}, v_1, 2, \texttt{[0]})} * \texttt{TREE}(\texttt{N}, v_2, 2, \texttt{[0]}) *$
$\forall\ v_3 \in \texttt{TreeRecords}(v_1).$
$[\texttt{nth}(\texttt{find}(v_1, v_3), 2)\ \texttt{inTree}\ v_0] *$
$\forall\ v_3 \in \texttt{TreeRecords}(v_2).$
$[\texttt{nth}(\texttt{find}(v_2, v_3), 2)\ \texttt{inTree}\ v_0]\}$
$\texttt{T} := *(\texttt{I} + 1);$
...
{?1234}

$\exists v_0 \exists v_1 \exists v_2 \exists v_3 \exists v_4$
$\boxed{[I + 1 \mapsto v_1 * I \mapsto nth(v_0, 0)] * \texttt{TREE}(nth(v_0, 0), nth([I, v_0, v_1], 1), 2, [0])} *$
$[Tmp\_r = 0 \vee Tmp\_r \in \texttt{TreeRecords}(v_0)] *$
$[nth(nth(find(v_2, T)), 2), 0) = (Tmp\_r)] *$
$[nth(nth(find(v_2, T)), 1), 0) = 0] *$
$[T \in \texttt{TreeRecords}(v_2)] *$
$P + 0 \mapsto N * P + 1 \mapsto T * [T \neq 0] *$
$\texttt{TREE}(\texttt{R}, v_2, 2, \texttt{[0,1]}) * \boxed{\texttt{Empty}} * \texttt{TREE}(\texttt{N}, v_4, 2, \texttt{[0]}) *$
$\forall\ v_5 \in \texttt{TreeRecords}(\boxed{[I, v_0, v_1]}).$
$[\texttt{nth}(\texttt{find}(\boxed{[I, v_0, v_1]}, v_5), 2)\ \texttt{inTree}\ v_2] *$
$\forall\ v_5 \in \texttt{TreeRecords}(v_4).$
$[\texttt{nth}(\texttt{find}(v_4, v_5), 2)\ \texttt{inTree}\ v_2]\}$
$\texttt{T} := *(\texttt{I} + 1);$
...
{?1234}

$\exists v_0 \exists v_1 \exists v_2 \exists v_3 \exists v_4 \exists v_5 \boxed{[T = v_2] *}$
$[I + 1 \mapsto v_2 * I \mapsto nth(v_1, 0)] * \texttt{TREE}(nth(v_1, 0), nth([I, v_1, v_2], 1), 2, [0]) *$
$[Tmp\_r = 0 \vee Tmp\_r \in \texttt{TreeRecords}(v_1)] *$
$[nth(nth(find(v_3, T)), 2), 0) = (Tmp\_r)] *$
$[nth(nth(find(v_3, T)), 1), 0) = 0] *$
$[T \in \texttt{TreeRecords}(v_3)] *$
$P + 0 \mapsto N * P + 1 \mapsto T * [T \neq 0] *$
$\texttt{TREE}(\texttt{R}, v_3, 2, \texttt{[0,1]}) * \texttt{Empty} * \texttt{TREE}(\texttt{N}, v_5, 2, \texttt{[0]}) *$
$\forall\ v_6 \in \texttt{TreeRecords}([I, v_1, v_2]).$
$[\texttt{nth}(\texttt{find}([I, v_1, v_2], v_6), 2)\ \texttt{inTree}\ v_3] *$
$\forall\ v_6 \in \texttt{TreeRecords}(v_5).$
$[\texttt{nth}(\texttt{find}(v_5, v_6), 2)\ \texttt{inTree}\ v_3]\}$
...
{?1234}

Fig. 5. Unfolding. Before forward propagating over the statement `T := *(I+1)`, we must first unfold the `TREE` rooted at I. This is the transition from the first to second goal above. The transition from the second to the third is the actual forward propagation.

$$
\begin{array}{l}
\texttt{Fixpoint}\ eval\,(env : id \to nat)\ (e : expr) := \\
\quad match\ e\ with \\
\qquad |\texttt{Const}\ c \Longrightarrow c \\
\qquad |\texttt{Var}\ v \Longrightarrow env\ v \\
\qquad |\texttt{Plus}\ e_1\ e_2 \Longrightarrow (eval\ env\ e_1) + (eval\ env\ e_2) \\
\qquad |\texttt{Minus}\ e_1\ e_2 \Longrightarrow (eval\ env\ e_1) - (eval\ env\ e_2) \\
\qquad |\texttt{Times}\ e_1\ e_2 \Longrightarrow (eval\ env\ e_1) * (eval\ env\ e_2)
\end{array}
$$

Now the interesting piece of the deep model is the construction of a *simplify* function. This function is defined and contains many of the useful simplifications explicitly. As an example, this function may have a "collect like terms" simplification the cancels out the $+b$ and $-b$. The simplification function can contain many complex algebraic simplifications. One then verifies the soundness of this function by proving a theorem of the form

$$\forall env. \forall e. eval\ env\ (simplify\ e) = eval\ env\ e$$

Once this is done in the infrastructure, the simplify function can be used for many different verifications. Finally, one can prove the original theorem by proving the following:

$$eval\ env\ (\texttt{Plus}\ a\ b) = eval\ env\ (Plus\ a\ (Plus\ b\ (Plus\ (Minus\ c\ b))))$$

The above theorem is proven by proving

$$\texttt{Plus}\ a\ b = simplify\ (Plus\ a\ (Plus\ b\ (Plus\ (Minus\ c\ b))))$$

and then combining this result with the theorem for *simplify*.

There is the disadvantage that a deep model is less flexible. In the shallow model, one can add complex specifications by simply defining new functions and properties as part of the proof. With the deep model, one actually has to make changes to the data structures. 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. As an example, one could replace the `Plus`, `Minus` and `Times` cases and create a data structure like the following:

$$\texttt{type}\ expr = \texttt{Const}\ int \mid \texttt{Var}\ id \mid \texttt{Fun}\ id \times list\ expr$$

The `Fun` term at the end replaces the `Plus`, `Minus` and `Times` terms from before. *id* is used to identify the function. One can then create different versions of *eval* that define different semantics even though the data structure itself does not change. This makes it possible to extend the infrastructure by creating variants of *eval*. It turns out by doing this, theorems in the infrastructure can be reused when new functions are added.

5 Related research

Our theorem proving framework is built using the Coq theorem prover and uses concepts from separation logic. Figure 6 show a number of key research projects leading up to our system[1, 4, 5, 7, 8, 10, 12, 13, 15].

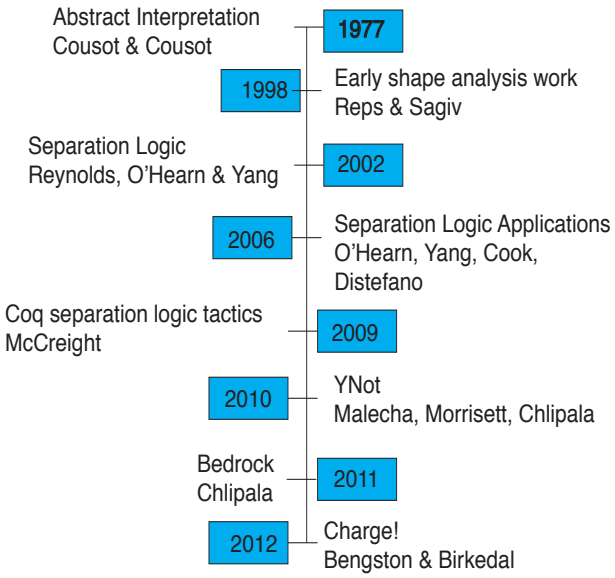


Fig. 6. A genealogy of some important past research contributions

Our PEDANTIC (Proof Engine for Deductive Automation using Non-deterministic Traversal of Instruction Code) framework is most similar to three other separation logic frameworks, Bedrock[4], Charge![1] and Ynot[10]. The primary issue with these three systems is that they only work on small examples. Their limit is a program of 20-30 lines.

The main contribution of our work is to find ways to increase the size and complexity of the programs that can be handled. Extending to larger examples has required many extensions and changes when compared to the other systems:

– **Deep model** We use a deep model to gain greater control over the reasoning processes within the theorem prover. Our main contribution in using the deep model is that of the design of the high level tactics. Many systems use deep models. For example, recent work by [9] incorporates a deep model into Bedrock using a technique called reflection which allows one to easily switch between a deep and shallow model. The have one key reification step in which they introduced some ML code to convert an expression from a shallow to deep representation. We found this not to be necessary. Moving

from deep to shallow can be done without ML code and our framework is setup so that this is the only direction that is needed.

- **Functional representation** The Btree example in [10] 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. This functional representation is the second parameter to the `Tree` predicate described above. 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** While merge algorithms have been commonly used in other static analysis systems[5], they have not been used in Coq separation logic frameworks. We are developing techniques to do merging within the context of separation logic. The other frameworks don't incorporate merge, they work by enumerating all execution paths and proving them separately. This works for small programs but for larger programs, the number of paths can explode.

6 Verification of the DPLL procedure

DPLL is an algorithm for determining whether a sentential logic formula in conjunctive normal form can be satisfied. The goal of doing this verification is to test out our framework on a data structure with a fairly complex invariant. This procedure can be coded with a limited number of lines of code yet the invariant is still reasonably complex and represents the kinds of stuff that arise in many larger programs.

We have developed an invariant for the DPLL procedure. It captures a formal description of the data structures used to implement the two variable algorithm[6]. We developed a C program implementing a simplified version of DPLL. The C program is about 250 lines. The invariant in our framework is coded in around 55 lines. The invariant requires a number of extensions to the basic PEDANTIC framework. However, these extensions follow nicely from the basic concepts already in the framework. As an example, there is a global variable that rather than pointing to a single linked list, points to an array each of whose elements is a pointer to the head of a linked list. We needed to add a construct to PEDANTIC first for representing arrays and second, we added a `foreach` construct which quantifies a variable over all array locations. This predicate then combines the evaluations of a predicate for each of the values with the separation *. This allows us to manage structures that have a variable number of children.

7 Proposed research for PhD

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 2 is complete, but correctness proofs of many auxiliary Lemmas still need to be completed and are for now `admitt`ed.

The research proposed for the PhD is to 1) complete the lemmas for the simple proof, 2) Finish up the DPLL verification and 3) to develop a front end UI (possibly in Python) to make the proofs more readable.

Acknowledgements The author would like to thank Scott Smith for his feedback.

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. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
3. Adam Chlipala. Certified programming with dependent types. 2009.
4. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd Programming Language Design and Implementation (PLDI)*, 2011.
5. 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.
6. 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.
7. Peter W. O'Hearn Dino Distefano and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
8. 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.
9. Gregory Malecha, Adam Chlipala, Thomas Braibant, Patrick Hulin, and Edward Z. Yang. Mirrorshard: Proof by computational reflection with verified hints. *CoRR*, abs/1305.6543, 2013.
10. Gregory Malecha and Greg Morrisett. Mechanized verification with sharing. In *ICTAC*, 2010.
11. Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Information and Media Technologies*, 4(2):304–316, 2009.
12. 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.
13. 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.
14. Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations*. 2011.
15. Reinhard Wilhel Shmuel Sagiv, Thomas W. Reps. Solving shape-analysis problems in languages with destructive updating. 20(1):1–50, 1998.