Using the Coq theorem prover to verify complex data structure invariants

Kenneth Roe
Department of Computer Science
The Johns Hopkins University
Email: roe@cs.jhu.edu

Scott Smith
Department of Computer Science
The Johns Hopkins University
Email: scott@jhu.edu

***Abstract*—While automated static analysis tools can find many useful software bugs, there are many important 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. Additionally, often the invariants are not well documented. However, maintaining them is necessary for correct operation of the software. 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 (though the correctness proofs of some of the tactics is incomplete). 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.

I. Introduction

Many software bugs can be traced to data structure invariant violations. These invariants can be quite complex. They are created as part of the process of writing a program. As an 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 (Proof Engine for Deductive Automation using Non-deterministic Traversal of Instruction Code) framework is implemented in Coq and is based on ideas from many other separation logic frameworks such as [8], [4], [20]. 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 [31]. 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 have made the decision to require the user to enter all loop invariants.

Forward propagation of an assertion over a block of statements can be automated. At the end of the block, often 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.

```
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

A. Overview of PEDANTIC contributions

In order to create an easy to use yet extensible framework, we took the work of [8], [4], [20], [25], [3] and introduced a number of 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 at the end of an if-then-else construct; and (4) use of simplification after other operations to put assertions into a canonical form.

1) 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 invariants. We then created a number of Gallina functions to manipulate these data structures. This gives us greater flexibility in 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 has to be regenerated in its application.

2) 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.

3) 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.

4) 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.

5) Framework extension: While our framework provides a number of useful constructs and tactics for reasoning about programs, we anticipate that many programs will require customized constructs. Programming styles vary greatly from system to system and trying to design generic constructs for all systems is impossible. Our framework allows for extensibility. Theorems and proofs in the framework are parameterized so that when new constructs are added, only simple proofs of certain properties are needed to verify the integrity of the extended framework. For example, our framework currently provides a `Tree` construct for linked lists or trees. This construct is parameterized to allow for a few basic variants. While the `Tree` construct works well for many data structures, there are many cases where it may not work well. For example a linked list with nodes of different types or a graph data structure in which multiple nodes point to the same child. For the former, the linked list may contain records with are either the name of a person or business. Depending on the type, the record may be of a different size. The `Tree` construct in the framework assumes all records are of the same size. However, the framework is setup so that one could easily add a `Tree2` construct (which is a slight variation of `Tree`) which handles the different sized records.

PEDANTIC is designed to be extensible. We use a deep encoding for defining abstract program states in Coq, which allows use of Galina functions to transform our assertions. The data structure type declarations for abstract states has a generic form for function calls and predicates. This allows us to add new functions and predicates without changing the data structure. We have designed our proofs so that they can be reused as new constructs are added. For example, if the generic tree predicate does not quite fit for a particular verification, a new predicate can be created and added to the system without needing to make major changes to the system's infrastructure.

II. A SAMPLE PROGRAM AND ITS INVARIANT

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

```
Inductive absExp : Type :=
 | AbsConstVal : (@Value unit) -> absExp
 | AbsVar : id -> absExp
 | AbsQVar : absVar -> absExp
 | AbsFun : id -> list absExp -> absExp.

Inductive absState :=
 | AbsExists : absExp -> absState -> absState
 | AbsAll : absExp ev eq f -> absState -> absState
 | AbsCompose : absState -> absState -> absState
 | AbsEmpty : absState
 | AbsLeaf : id -> (list absExp) -> absState
 | AbsMagicWand : absState -> absState -> absState
 | AbsUpdateVar : id -> absExp -> absState ->absState
 | AbsUpdateLoc : absExp -> absExp -> absState-> absState
 | AbsUpdateWithLoc : id -> absExp -> absState-> absState
 | AbsUpdState : absState-> absState -> absState -> absState
 ...
```

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

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.

A. Separation Invariants

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

Figure 2 shows the Coq definitions used for our state assertions. 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.

Shallow embeddings such as Bedrock[8] 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 have generic function (`AbsFun`) and predicate (`AbsLeaf`) constructs in the separation logic grammar. The semantics provide a predefined set of functions and predicates. However, new functions and predicates can easily be added as the separation logic data structure does not need to be modified.

The `absFun` and `absLeaf` datatypes are given semantics by the Coq functions `basicEval` and `basicState`. The `basicState` function defines three types of leaf predicates, `AbsPredicate`, `AbsCell` and `AbsTree`.

$(e,h) \vdash$ `AbsCompose` $s_1 s_2$ iff $\exists h_1 h_2, dom(h_1) \cap dom(h_2) = \emptyset \wedge$ $(e,h_1) \vdash s_1 \wedge (e,h_2) \vdash s_2 \wedge h = h_1 \cup h_2$

$(e,h) \vdash$ `AbsMagicWand` $s_1 s_2$ iff $\exists h_1 h_2, dom(h) \cap dom(h_2) = \emptyset \wedge$ $(e,h_1) \vdash s_1 \wedge (e,h_2) \vdash s_2 \wedge h_1 = h \cup h_2$

$(e,h) \vdash$ `AbsEmpty` iff $dom(h) = \emptyset$

$(e,h) \vdash$ `AbsExists` $v.s$ iff $\exists x \in \mathcal{V}.(e,h) \vdash s[x/v]$

$(e,h) \vdash$ `AbsAll` $v \in R(\bar{p}).s$ iff $\forall x \in \mathcal{R}(R(\ll \bar{p} \gg e))$

$(e,h) \vdash$ `AbsUpdateVar` i v s iff $(e[i] = \ll v \gg e \wedge \exists x \in \mathcal{V}.(e[x/i], h) \vdash s$

$(e,h) \vdash$ `AbsUpdateWithLoc` i v s iff $(e[i] = h(\ll v \gg e) \wedge \exists x \in \mathcal{V}.(e[x/i], h) \vdash s$

$(e,h) \vdash$ `AbsUpdateLoc` l v s iff $(\ll v \gg e = h(\ll l \gg e) \wedge$ $\exists x \in \mathcal{V}.(e, h[x/ \ll l \gg e]) \vdash s$

$(e,h) \vdash$ `AbsUpdState` s_1 s_2 s_3 iff $(e,h) \vdash$ `AbsCompose` (`AbMagicWand` s_1 s_2) s_3

$(e,h) \vdash [P]$ iff $\ll P \gg e \neq 0$ and $dom(h) = \emptyset$

$(e,h) \vdash l \mapsto v$ iff $h(\ll l \gg e) = \ll v \gg e \wedge$ $\forall x \in \mathcal{N}, x \neq \ll l \gg e \rightarrow x \notin dom(h)$

$(e,h) \vdash$ `TREE`$(r,v,n,\bar{f})$ iff $(\ll r \gg e = 0 \wedge h = \emptyset) \vee$
$\exists h_0, \ldots, h_{n+m}, x_0, \ldots, x_{n-1}, v_0, \ldots, v_m. \ll r \gg e \neq 0 \wedge$
$(e,h_0) \vdash (\ll r \gg e + 0) \mapsto x_0 \ \wedge \ldots \wedge (e,h_{n-1}) \vdash (\ll r \gg e + n - 1) \mapsto x_{n-1} \wedge$
$(e,h_n) \vdash$ `TREE`$(h(\ll r \gg +f_0), v_{f_0}, s, \bar{f}) \ \wedge \ldots \wedge$
$(e,h_{n+m-1}) \vdash$ `TREE`$(h(\ll r \gg +f_{m-1}), v_{f_{m-1}}, s, \bar{f}) \wedge$
`combine`$(h, h_0, \ldots, h_{n+m-1}) \wedge$
$v = [\ \ll r \gg e, v_0, \ldots, v_{n-1}\ ] \wedge \forall i < n.i \notin \bar{f}$ implies $v_i = x_i$
where `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 three constructs defined by `basicState` for `absLeaf`. (e,h) represents a concrete state. $(e,h) \vdash s$ means that (e,h) satisfies the assertion s. $e \in$ `Id` $\rightarrow$ `Nat` represents assignments for program variables and $h =$ `Nat` $\rightarrow$ `option Nat` 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.

`AbsLeaf AbsPredicate` P, denoted $[P]$, indicates an arbitrary predicate on the heap, `AbsLeafAbsCell` 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 predicates and some of the core separation logic are given in Figure 3.

B. Expressions and values

The values in our logic can be either integers or lists of values as represented by the following Coq data type:

```
Inductive Value :=
     | NatValue : Int -> Value
     | ListValue : (list Value) -> Value.
```

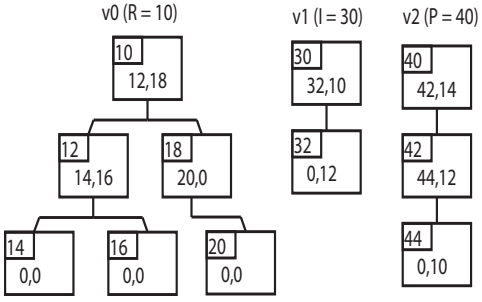
Expressions in our system consist of basic operators on the value structure as well as specialized operations to facilitate our logic. The semantics of these operators are given in figure 4. A more detailed explanation of the latter is given in the next section where we outline the verification of our tree traversal invariant.

C. An example state assertion

To better understand state assertions we present a sample heap snapshot of our example program in Figure 5. This example shows the heap memory after two loop iterations have

$$
\begin{array}{lcll}
\ll c \gg e\ b & = & c & \text{constant} \\
\ll v \gg e\ b & = & e[v] & \text{environment variable} \\
\ll v \gg e\ b & = & b[v] & \text{quantifier variable} \\
\ll l_1 + l_2 \gg e\ b & = & \texttt{NatValue}\ v_1 + v_2 & \text{iff } \texttt{NatValue} v_1 = \ll l_1 \gg e\ b \wedge \\
 & & & \texttt{NatValue}\ v_2 = \ll l_2 \gg e\ b \\
 & & \texttt{NoValue} & \text{otherwise} \\
\ll l_1 * l_2 \gg e\ b & = & \texttt{NatValue}\ v_1 * v_2 & \text{iff } \texttt{NatValue}\ v_1 = \ll l_1 \gg e\ b \wedge \\
 & & & \texttt{NatValue}\ v_2 = \ll l_2 \gg e\ b \\
 & & \texttt{NoValue} & \text{otherwise} \\
\ll l_1 = l_2 \gg e\ b & = & \texttt{NatValue}\ 1 & \text{iff } \ll l_1 \gg e\ b = \ll l_2 \gg e\ b \\
 & & \texttt{NatValue}\ 0 & \text{otherwise} \\
\ll l_1 < l_2 \gg e\ b & = & \texttt{NatValue}\ 1 & \text{iff } \texttt{NatValue}\ v_1 = \ll l_1 \gg e\ b \wedge \\
 & & & \texttt{NatValue}\ v_2 = \ll l_2 \gg e\ b \wedge v_1 < v_2 \\
 & & \texttt{NoValue} & \text{otherwise} \\
 & & \texttt{NatValue}\ 0 & \text{iff } \texttt{NatValue}\ v_1 = \ll l_1 \gg e\ b \wedge \\
 & & & \texttt{NatValue}\ v_2 = \ll l_2 \gg e\ b \wedge v_1 \not< v_2 \\
 & & \texttt{NoValue} & \text{otherwise} \\
\ldots & & & \\
\ll \texttt{find}(l, v) \gg \texttt{e b} & = & F(\ll l \gg e\ b, \ll v \gg e\ b) & \\
\text{where} & & F(l, v) = v & \text{iff } v = \texttt{ListValue}[l, \ldots] \\
 & & F(l, v) = v_i & \text{iff } v = \texttt{ListValue}[l', v_1, \ldots, v_n] \wedge \\
 & & & v_i = \texttt{ListValue}[l, \ldots] \wedge \\
 & & & i \in [1, \ldots, n] \\
 & & F(l, v) = \texttt{NoValue} & \text{otherwise} \\
\ll \texttt{rangeSet}(f) \gg \texttt{e b} & = & RS(\ll f \gg e\ b) & \\
\text{where} & & RS(v) = l + + RS(v_1) + + \ldots + + RS(v_n) & \text{iff } v = \texttt{ListValue}[l, \ldots] \\
 & & RS(v) = [] & \text{otherwise} \\
\ll \texttt{rangeNumeric}(s, f) \gg \texttt{e b} & = & RN(st, en) & \text{iff } \ll s \gg e = \texttt{NatValue}\ st \wedge \\
 & & & \ll f \gg e\ b = \texttt{NatValue}\ en \wedge \\
\text{where} & & RS(v) = [st, \ldots, en] & \text{iff } v = \texttt{ListValue}[l, \ldots]
\end{array}
$$

Fig. 4. Semantics for expressions. The first three cases are for the `AbsExp` core and the rest are for some of the functions embedded in `absFun`.



$\texttt{AbsExists}\ v_3.\texttt{AbsExists}\ v_2.\texttt{AbsExists}\ v_1.$
$\quad \texttt{TREE}(\texttt{R}, v_3, 2, [0, 1]) *$
$\quad \texttt{TREE}(\texttt{I}, v_2, 2, [0]) *$
$\quad \texttt{TREE}(\texttt{P}, v_1, 2, [0]) *$
$\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_2).$
$\quad\quad [\texttt{nth}(\texttt{find}(v_2, v_0), 2)\ \texttt{inTree}\ v_3] *$
$\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_1).$
$\quad\quad [\texttt{nth}(\texttt{find}(v_1, v_0), 2)\ \texttt{inTree}\ v_3] *$
$\quad\quad [\texttt{T} = 0 \vee \texttt{T}\ \texttt{inTree}\ v_3]$

Fig. 5. 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.

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 [20]. 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 $x \in \texttt{TreeRecords}(y)$, 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, $\texttt{find}([30, [32, [0], 12], 10], 32)$ will return $[32, [0], 12]$. Finally, the last line states that either `T` is `0` or that `T` points to an element in `R`.

III. Verifying the assertions

We now show how to verify that the invariant of figure 5 holds throughout our program. Figure 9 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. The comments show the program that has been converted to Coq notation. One starts by forward propagating the invariant (called `afterInitAssigns` here). When we hit the while loop, we need to supply a loop invariant (called `loopInv` here). In this case, the invariant turns out to be exactly the same as the one given in figure 5.

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`

$$\frac{}{\{s\}\ \texttt{x} := \texttt{e}\ \{\texttt{absUpdateVar}\ x\ e\ s\}\ d}\ \text{assignment}$$

$$\frac{\mathit{canSave}(e+f, s)}{\{s\}\ \texttt{x} := \texttt{e}\ \texttt{->}\ \texttt{f}\ \{\texttt{AbsUpdateWithLoc}\ x\ (e+f)\ s\}}\ \text{load}$$
where $\mathit{canSave}(l, s) = \forall eh.(e, h) \vdash s \rightarrow h(l) \neq \texttt{None}$

$$\frac{\mathit{canSave}(x+f, s)}{\{s\}\ \texttt{x}\ \texttt{->}\ \texttt{f} := \texttt{e}\ \{\texttt{AbsUpdateLoc}\ l\ x\ s\}}\ \text{store}$$

$$\frac{\begin{array}{c} l = e \qquad \mathit{validExp}(e, s) \qquad \texttt{n} : \mathit{nat} \\ \exists st.st \vdash s * - \qquad (\exists v_0, \ldots, v_{n-1}.l \mapsto v_0, \ldots l + n - 1 \mapsto v_{n-1}) \end{array}}{\begin{array}{c} \{s\}\ \texttt{DELETE e, n}\ \{s * -(\exists v_0, \ldots, v_{n-1}.l \mapsto v_0, \\ \ldots, l + n - 1 \mapsto v_{n-1})\} \end{array}}\ \text{del}$$

$$\frac{\texttt{n} : \mathit{nat}}{\{s\}\ \texttt{v} := \texttt{NEW n}\ \{s * \texttt{v} \mapsto c_0 * \ldots * \texttt{v} + \texttt{n} - 1 \mapsto c_{\texttt{n}-1}\}}\ \text{new}$$

Fig. 6. Forward propagation rules. The inference rules above are used to propagate over the statements of a program.

$$\frac{\texttt{absUpdateVar}\ v\ e\ s}{\texttt{absExists}\ x.s[x/v] * [v = e]}\ \text{absUpdateVar elim}$$

$$\frac{\texttt{absUpdateWithLoc}\ v\ e\ s}{\texttt{AsExists}\ x.(s[x/v] * [v = q[x/v]])}\ \text{absUpdateWithLoc elim1}$$
where $s = \ldots * e \mapsto q * \ldots$

$$\frac{\texttt{absUpdateWithLoc}\ v\ e\ s}{\begin{array}{c} \texttt{absUpdateWithLoc}\ v\ e\ s \\ [\texttt{absExists}\ x.ss[x/v]/\texttt{absUpdateWithLoc}\ v\ e'\ ss] \end{array}}\ \text{elim2}$$
where $s = \ldots * \texttt{absUpdateWithLoc}\ v\ e'\ ss * \ldots$

$$\frac{\texttt{asUpdateWithLoc}\ v\ e\ s}{s'}\ \text{absUpdateWithLoc elim3}$$
where s' is s with all predicates involving e removed. This rule requires that v only be used inside of predicates of the form $[\ldots]$ within s.

$$\frac{\texttt{absUpdateLoc}\ l\ e\ s}{s[l \mapsto e/l \mapsto q]}\ \text{absUpdateLoc elim}$$
where $s = \ldots * l \mapsto q * \ldots$

Fig. 7. Rules for eliminating update constructs

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 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.

$$\frac{s * -(\texttt{absExists}\ v.l_1 \mapsto v) * \ldots * (\texttt{absExists}\ v.l_n \mapsto v)}{s[\texttt{AbsEmpty}/(l_1 \mapsto e_1, \ldots, l_n \mapsto e_n)]}$$
where $s = \ldots l_1 \mapsto e_1, \ldots, l_n \mapsto e_n \ldots$

Fig. 8. Rule for eliminating magic wand

```
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.

        (* (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.
```

Fig. 9. Top level Coq proof of our program. Notice how the structure of the proof matches the structure of the program

A. Forward propagation

Figure 6 shows the rules for forward propagating over a state. We have set up our framework so that these rules can be applied mechanically. Notice that the assertion before the statement in each rule does not need to be broken apart. We use absUpdateVar, absUpdateLoc and absUpdateWithLoc to represent updates. To show how this propagation works, below is shown the state before the assignment N ::=!P in figure 9:

$$
\begin{array}{l}
[\,\mathtt{T}=0]* \\
\mathtt{AbsExists}\ v_3.\mathtt{AbsExists}\ v_2.\mathtt{AbsExists}\ v_1. \\
\quad \mathtt{TREE}(\mathtt{R}, v_3, 2, [0,1])* \\
\quad \mathtt{TREE}(\mathtt{I}, v_2, 2, [0])* \\
\quad \mathtt{TREE}(\mathtt{P}, v_1, 2, [0])* \\
\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2). \\
\qquad [\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_1). \\
\qquad [\mathtt{nth}(\mathtt{find}(v_1, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\quad [\mathtt{T}=0 \vee \mathtt{T}\ \mathtt{inTree}\ v_3]
\end{array}
$$

After the assignment, we get the state shown below:

$$
\begin{array}{l}
\mathtt{AbsUpdateVar}\ \mathtt{N}\ \mathtt{P} \\
\quad ([\,\mathtt{T}=0]* \\
\quad \mathtt{AbsExists} v_3.\mathtt{AbsExists} v_2.\mathtt{AbsExists} v_1. \\
\qquad \mathtt{TREE}(\mathtt{R}, v_3, 2, [0,1])* \\
\quad \mathtt{TREE}(\mathtt{I}, v_2, 2, [0])* \\
\qquad \mathtt{TREE}(\mathtt{P}, v_1, 2, [0])* \\
\qquad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2). \\
\qquad\quad [\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\qquad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_1). \\
\qquad\quad [\mathtt{nth}(\mathtt{find}(v_1, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\qquad [\mathtt{T}=0 \vee \mathtt{T}\ \mathtt{inTree}\ v_3])
\end{array}
$$

We can then propagate over the next statement, NEW P, ANum(Size_l) and we get the state shown below:

$$
\begin{array}{l}
\mathtt{AbsExists}\ v_7.\mathtt{AbsExists}\ v_6.\mathtt{AbsExists}\ v_5.\mathtt{AbsExists}\ v_4. \\
\quad v_4+1 \mapsto v_6 * v+4 \mapsto v_5 * \mathtt{P}=v_4* \\
\quad \mathtt{AbsUpdateVar}\ \mathtt{N}\ v_7 \\
\qquad ([\,\mathtt{T}=0]* \\
\qquad \mathtt{AbsExists} v_3.\mathtt{AbsExists} v_2.\mathtt{AbsExists} v_1. \\
\qquad\quad \mathtt{TREE}(\mathtt{R}, v_3, 2, [0,1])* \\
\qquad\quad \mathtt{TREE}(\mathtt{I}, v_2, 2, [0])* \\
\qquad\quad \mathtt{TREE}(\mathtt{P}, v_1, 2, [0])* \\
\qquad\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2). \\
\qquad\qquad [\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\qquad\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_1). \\
\qquad\qquad [\mathtt{nth}(\mathtt{find}(v_1, v_0), 2)\ \mathtt{inTree}\ v_3]* \\
\qquad\quad [\mathtt{T}=0 \vee \mathtt{T}\ \mathtt{inTree}\ v_3])
\end{array}
$$

B. Merging states (+fold/unfold and simplification)

Shown below are the two states from the ends of then- and else- blocks of the if statement when mergeTheorem1 of figure 9 is applied. These are the states generated by the forward propagation of the two branches. They are both similar as they both were derived from the same root invariant. The merge process involves several steps. Hence, this merge example also illustrates simplification, unfolding and folding.

$$
\begin{array}{l}
\mathtt{AbsUpdateVar}\ \mathtt{T}\ 0\ ([\mathtt{I}=0]*[\mathtt{tmp\_l}=0 \wedge \mathtt{tmp\_r}=0]* \\
\quad (\mathtt{AbsUpdateWithLoc}\ \mathtt{tmp\_r}\ (\mathtt{T}+1) \\
\qquad (\mathtt{AbsUpdateWithLoc}\ \mathtt{tmp\_l}\ (\mathtt{T}+0) \\
\qquad\quad (\mathtt{AbsUpdateLoc}\ \mathtt{P}\ N\ (\mathtt{AbsUpdateLoc}\ (\mathtt{P}+1)\ \mathtt{T} \\
\qquad\qquad (\mathtt{AbsExists}\ v_6\ (\mathtt{AbsExists}\ v_5\ (\mathtt{AbsExists}\ v_4 \\
\qquad\qquad\quad (\mathtt{AbsUpdateVar}\ \mathtt{N}\ v_2\ (\\
\qquad\qquad\quad \mathtt{P}+1 \mapsto v_6 * \mathtt{P} \mapsto v_5 * [\mathtt{P}=v_4] * [0<\mathtt{T}]* \\
\qquad\qquad\quad (\mathtt{AbsExists}\ v_3\ (\mathtt{AbsExists}\ v_2\ (\mathtt{AbsExists}\ v_1 \\
\qquad\qquad\qquad \mathtt{TREE}(\mathtt{R}, v_1, 2, [0,1])* \\
\qquad\qquad\qquad \mathtt{TREE}(\mathtt{I}, v_2, 2, [0])* \\
\qquad\qquad\qquad \mathtt{TREE}(\mathtt{P}, v_3, 2, [0])* \\
\qquad\qquad\qquad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2). \\
\qquad\qquad\qquad\quad [\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_1]* \\
\qquad\qquad\qquad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_3). \\
\qquad\qquad\qquad\quad [\mathtt{nth}(\mathtt{find}(v_3, v_0), 2)\ \mathtt{inTree}\ v_1]* \\
\qquad\qquad\qquad [\mathtt{T}\ \mathtt{inTree}\ v_1]))))))))))))))
\end{array}
$$

$$
\begin{array}{l}
\mathtt{AbsUpdateVar}\ \mathtt{I}\ \mathtt{tmp\_l} \\
\quad (\mathtt{AbsUpdateWithLoc}\ \mathtt{tmp\_l}\ \mathtt{T}+0 \\
\qquad (\mathtt{AbsUpdateWithLoc}\ \mathtt{T}\ \mathtt{I}+1 \\
\qquad\quad ([\mathtt{I} \neq 0] * [\mathtt{tmp\_l}=0 \wedge \mathtt{tmp\_r}=0]* \\
\qquad\qquad (\mathtt{AbsUpdateWithLoc}\ \mathtt{tmp\_r}\ (\mathtt{T}+1) \\
\qquad\qquad\quad (\mathtt{AbsUpdateWithLoc}\ \mathtt{tmp\_l}\ (\mathtt{T}+0) \\
\qquad\qquad\qquad (\mathtt{AbsUpdateLoc}\ \mathtt{P}\ \mathtt{N}\ (\mathtt{AbsUpdateLoc}\ (\mathtt{P}+1)\ \mathtt{T} \\
\qquad\qquad\qquad\quad (\mathtt{AbsExists}\ v_6\ (\mathtt{AbsExists}\ v_5(\mathtt{AbsExists}\ v_4 \\
\qquad\qquad\qquad\qquad (\mathtt{AbsUpdateVar}\ \mathtt{N}\ v_2 \\
\qquad\qquad\qquad\qquad \mathtt{P}+1 \mapsto v_6 * \mathtt{P} \mapsto v_5 * [\mathtt{P}=v_4] * [0<\mathtt{T}]* \\
\qquad\qquad\qquad\qquad (\mathtt{AbsExists}\ v_3\ (\mathtt{AbsExists}\ v_2(\mathtt{AbsExists}\ v_1 \\
\qquad\qquad\qquad\qquad\quad \mathtt{TREE}(\mathtt{R}, v_1, 2, [0,1])* \\
\qquad\qquad\qquad\qquad\quad \mathtt{TREE}(\mathtt{I}, v_2, 2, [0])* \\
\qquad\qquad\qquad\qquad\quad \mathtt{TREE}(\mathtt{N}, v_3, 2, [0])* \\
\qquad\qquad\qquad\qquad\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2). \\
\qquad\qquad\qquad\qquad\qquad [\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_1]* \\
\qquad\qquad\qquad\qquad\quad \mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_3). \\
\qquad\qquad\qquad\qquad\qquad [\mathtt{nth}(\mathtt{find}(v_3, v_0), 2)\ \mathtt{inTree}\ v_1]* \\
\qquad\qquad\qquad\qquad [\mathtt{T}\ \mathtt{inTree}\ v_1]))))))))))))))) * - \\
(\mathtt{AbsExists}\ v_2\ (\mathtt{AbsExists}\ v_1(\mathtt{AbsExists}\ v_0 \\
\quad v_0+1 \mapsto v_2 * v_0+0 \mapsto v_1 * [\mathtt{I}=v_0])))
\end{array}
$$

We start by doing simplifications and transformations. The results are shown below. Rules for eliminating updateVar, updateLoc and updateWithLoc are shown in figure 7. There are many other simplification rules that correspond to simple tautologies. For example, the expression $x+0$ will e replaced with x. We do not enumerate these rules.

```
(AbsExists v6 (AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [T = 0] * [I = 0] * (AbsEmpty * AbsEmpty)*
    [N = v5] * (P + 1) ↦ v6 * P ↦ N * [P = v4] * [0 < v5]*
      TREE(R, v1, 2, [0, 1])*
      TREE(I, v2, 2, [0])*
      TREE(N, v3, 2, [0])*
     AbsAll v0 ∈ TreeRecords(v2).
        [nth(find(v2, v0), 2) inTree v1]*
     AbsAll v0 ∈ TreeRecords(v3).
        [nth(find(v3, v0), 2) inTree v1]*
      [v6 inTree v1])))))))
```

```
(AbsExists v6 (AbsExists v5(AbsExists v4
  (AbsExists v3 (AbsExists v2(AbsExists v1
    [I = tmp_l]*
     (AbsUpdateWithLoc tmp_l v6
       (AbsUpdateWithLoc T (v6 + 1)
         ([0 < v5] * [tmp_l = 0 ∧ tmp_r = 0]*
           (AbsUpdateWithLoc tmp_r (T + 1)
             (AbsUpdateWithLoc tmp_l T
              P + 1 ↦ T * P ↦ N * [P = v4] * [0 < T]*
                 TREE(R, v1, 2, [0, 1])*
                 TREE(I, v2, 2, [0])*
                 TREE(N, v3, 2, [0])*
                AbsAll v0 ∈ TreeRecords(v2).
                   [nth(find(v2, v0), 2) inTree v1]*
                AbsAll v0 ∈ TreeRecords(v3).
                   [nth(find(v3, v0), 2) inTree v1]*
                 [T inTree v1])))))) * −
  (AbsExists v1(AbsExists v0
   v6 + 1 ↦ v2 * v6 ↦ v1))))))))
```

On the right hand side, we need remove the magic wand. To do this, we first need to unfold the tree rooted at I. After unfolding, we simplify. The result of unfolding is shown below.

```
(AbsExists v8 (AbsExists v7
  (AbsExists v6 (AbsExists v5(AbsExists v4
    (AbsExists v3 (AbsExists v2(AbsExists v1
      (AbsUpdateWithLoc tmp_l v6
        (AbsUpdateWithLoc T (v6 + 1)
          (AbsUpdateWithLoc tmp_r (T + 1)
            (AbsUpdateWithLoc tmp_l T
             (P + 1 ↦ T * P ↦ N * [P = v4] * [0 < T]*
                TREE(R, v1, 2, [0, 1])*
                v6 + 1 ↦ v8 * v6 + 0 ↦ nth(v7, 0)*
                TREE(nth(v7, 0), nth(v6 :: v7 :: v8 :: nil, 1),
                  2, [0])*
                TREE(N, v3, 2, [0])*
                AbsAll v0 ∈
                   TreeRecords(v6 :: v7 :: v8 :: nil).
                  [nth(find(v6 :: v7 :: v8 :: nil, v0), 2)
                  inTree v1]*
                AbsAll v0 ∈ TreeRecords(v3).
                  [nth(find(v3, v0), 2) inTree v1]*
                [T inTree v1]) * [N = v5])]*
      [tmp_l = 0] * [tmp_r = 0] * [0 > v6]) * −
  (AbsExists v1(AbsExists v0
   v6 + 1 ↦ v2 * v6 ↦ v1))))))))))
```

We now use the rule shown in figure 8 to eliminate the magic wand construct.

```
(AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [I = nth(v5, 0)] * (P + 1) ↦ v1 * P ↦ N * [0 < v1]*
    TREE(R, v2, 2, [0, 1])*
    TREE(I, v5, 2, [0])*
    TREE(N, v3, 2, [0])*
    [TinTreev2]*
     AbsAll v0 ∈ TreeRecords(v5).
        [nth(find(v5, v0), 2) inTree v2]*
     AbsAll v0 ∈ TreeRecords(v3).
        [nth(find(v3, v0), 2) inTree v2]*
      [v1 inTree v2])))))
```

We then merge the two states. The result is shown below. Most of the work can be done automatically by pairing off identical components in the two branches. There is one term, $[\mathtt{T} = 0 \vee \mathtt{T}\ \mathtt{inTree}\ v_1]$, that the user needs to introduce. It is derived from the fact that that $[\mathtt{T}\ \mathtt{inTree}\ v_1]$ is in the left state and $[\mathtt{T} = 0]$ is in the right state. They can be put together in a disjunction.

```
(AbsExists v6 (AbsExists v5 (AbsExists v4
  (AbsExists v3 (AbsExists v2 (AbsExists v1
    [I = I] * (P + 1) ↦ v6 * P ↦ N * [0 < v5]*
    TREE(R, v1, 2, [0, 1])*
    TREE(I, v2, 2, [0])*
    TREE(N, v3, 2, [0])*
    AbsAll v0 ∈ TreeRecords(v5).
      [nth(find(v5, v0), 2) inTree v2]*
    AbsAll v0 ∈ TreeRecords(v3).
      [nth(find(v3, v0), 2) inTree v2]*
    [v6 inTree v2]*
    [T = 0 ∨ T inTree v1])))))))
```

As a final step, we want to fold back the tree rooted at N with the cells at locations P and T + 1. The result is shown below.

```
(AbsExists v3 (AbsExists v2 (AbsExists v4
  AbsAll v0 ∈ TreeRecords(v3).
    [nth(find(v3, v0), 2) inTree v1]*
  TREE(P, v3, 2, [0])*
  [nth(nth(v3, 1), 0) = N] * [0 < nth(v3), 2]*
  TREE(R, v1, 2, [0, 1])*
  TREE(I, v2, 2, [0])*
  AbsAll v0 ∈ TreeRecords(v2).
    [nth(find(v2, v0), 2) inTree v1]*
  AbsAll v0 ∈ TreeRecords(v3).
    [nth(find(v3, v0), 2) inTree v1]*
  [T = 0 ∨ T inTree v1])))
```

C. Entailment Proofs

There are three entailment proofs in our tree traversal example. The general methodology in PEDANTIC is to pair off identical components in the same manner as is done for merge and then to prove the remaining components separately.

IV. Coq source files

The main proof described in this paper with its 12 lemmas can be found in `TreeTraversal.v`. We also included the file `SatSolverMain.v` which shows the beginnings of our DPLL solver verification. We have also included `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.

V. 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].

The basic concepts were later introduced in the Coq theorem prover. McCreight developed a set of 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 frameworks for verifying imperative programs with recursive data structures. Coq's Ltac language for creating tactics is limited and hence limits the ability to do automation in any of these systems. PEDANTIC uses a deep model which gives greater flexibility in designing tactics.

In recent years, the VST system was 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.

VI. Conclusion

This paper has covered the basic concepts of 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 5 is complete, but correctness proofs of most auxiliary Lemmas still need to be completed and are for now `admitted`.

We are also working on the verification of the DPLL [11] based SAT solver invariant. After this work is done, the next project will be dealing with function calls. The idea is to use our invariant language to specify pre- and post- conditions which are an abstraction of the function's behavior and then to use those conditions in the verification of code that calls those functions. Separation logic provides some nice framing properties which we anticipate will be useful to generalize pre- and post- conditions after a function verification is done.

We are working towards 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 [31]. 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 straight forward and can be automated. For example, `treeRef1`, `treeRef2`, `storeCheck1` and `storeCheck2` turn out to be fairly simple proofs that can likely be automated (though this is not always the case). The areas of complexity are in doing implication proofs and merges.

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 for the tool to be practical. 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.
[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] Kenneth Roe and Scott Smith. Coqpie: An ide aimed at improving proof development productivity (rough diamond. In *ITP*, 2016.
[32] 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.
[33] Reinhard Wilhel Shmuel Sagiv, Thomas W. Reps. Solving shape-analysis problems in languages with destructive updating. 20(1):1–50, 1998.
[34] 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.
[35] Ting Zhang Zohar Manna, Henny B. Sipma. Verifying balanced trees. In *Proceedings of the Symposium on Logical Foundations of Computer Science (LFCS 2007)*, 2007.