Toward a tool to verify complex data structure invariants

by

Kenneth David Roe

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

February, 2018

Abstract

Formal methods provide the potential to create effective tools for finding bugs in software. However, the effectiveness of existing tools is limited: many existing static analysis tools provide limited abstractions, and interactive theorem proving techniques while providing deeper verifications are often too tedious to be practical.

The goal of our research is to make interactive theorem proving techniques more practical to use. We started by developing the PEDANTIC framework in Coq which is based on Separation logic to document and reason about program invariants. Initial experience with PEDANTIC in Coq showed such reasoning to be quite tedious: proving invariants correct was slow and cumbersome. So, we next developed two tools aimed at improving prover productivity. First we developed CoqPIE, an interactive IDE aimed at streamlining common proof development tasks such as replaying theorems when there are changes. Next we developed an advanced rewriting library which gives Coq more powerful expression simplification algorithms. To stress test PEDANTIC and these new tools, we culminated with a verification of data structure invariants of the DPLL satisfiability solver algorithm, a complex 200-line C program.

ABSTRACT

While the DPLL proof is not fully complete, key pieces of the proof are done and many important tactics and other building blocks were developed. Overall, the research presented gives a greater understanding of many proof development productivity issues and how to address these issues in practice.

Primary Reader: Scott F. Smith

Secondary Reader: Matt Green

Secondary Reader: ThanhVu Nguyen

Acknowledgments

I would first like to thank my advisor Scott Smith for all the time he spent helping me write this thesis, for reviewing all the papers I submitted for publication over the years and for helping me market my research to the interactive theorem proving community. I would also like to thank my other two committee members, Matt Green and ThanhVu Nguyen for their feedback on my research.

I would also like to thank all of the following people who gave me feedback on research papers or helped me out with Coq development during my graduate school years: Gregory Malecha, William Mansky, Adam Chlipala, Jesper Bengston, and Valentin Robert. Also, I would like to thank the Coq development team and other members of the Coq community who answered my questions on the Coq-Club mailing list.

Finally, I would like to thank my wife Ha for providing most of our family's financial support during the years I was in graduate school.

Dedication

This thesis is dedicated to Ha, Maya, Thanh and Leah

Contents

List of Figures

Chapter 1

Introduction

Finding and fixing bugs is often the hardest part of software development. And it is important: many security vulnerabilities can be tied to software bugs. Software development companies employ QA teams to test software before release, and often the cost of QA is as much as the original cost of software development. Despite this cost, QA still cannot find all bugs; after software is released, considerable effort is spent on finding and fixing bugs, some of which can cause considerable damage. For example, the Heart Bleed bug [6] was quite costly to many organizations hosting websites.

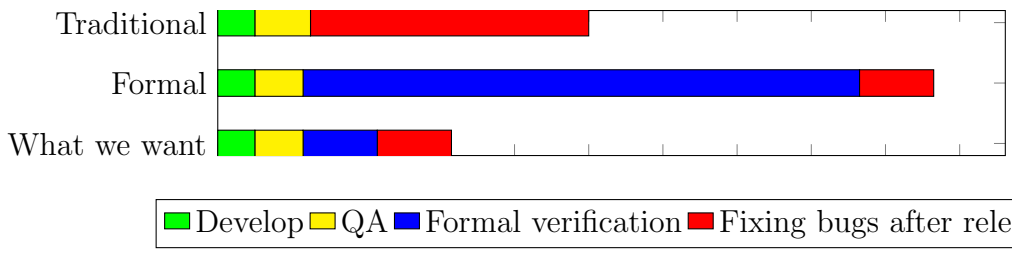
Many of these bugs can be difficult to find, because they only happen when a combination of conditions are true, and these conditions may occur at the customer site but not in the QA environment. Also many bugs infrequently arise, for example, they may only happen once a week in a software system that runs continuously.

Reproducing and understanding the conditions for these bugs is very challenging.

Formal verification can potentially provide tools that substantially improve the ability to find and fix bugs before software is released. The author's experience as a software engineer indicates that up to 90% of bugs (or more) could be found by formal methods . The Heart Bleed bug mentioned above is one example.

Existing efforts fall into two categories. First there are static analysis tools which do automated program analyses; the user does not need to produce any formal specification. The tool simply performs a symbolic execution and attempts to verify that common errors such as memory leaks and nil pointer references do not happen based on the information available from the symbolic execution. While such tools are useful, they generate many false positives since the analysis is very incomplete. The Coverity [20] analyzer is a such a tool; it can be run completely automatically and as such has gained commercial acceptance.

The second approach is to use an interactive theorem prover. This approach is too tedious to use on anything but small programs. However, one can produce and verify much more detailed specifications, detailed enough to find many useful bugs. VST [10] and Iris [49] are examples of frameworks for supporting program verification using the Coq interactive theorem prover. Both systems provide primitives for reasoning about programs using concurrent separation logic, and both are built on top of the Coq theorem prover. Iris is a new system which focuses on verifying multi-threaded programs; little work has been performed on verifying complex data



The chart above summarizes the costs of different methodologies. Development and QA are quite cheap in traditional development (without formal methods). However, the cost of fixing bugs after release is quite high. In addition to the cost of repairing bugs that customers find, one also needs to factor in the cost of software failures. The second bar shows formal methods. The problem here is that the cost of doing a verification is quite high and most likely more than fixing the bugs at the customer site. One key goal of this thesis is to bring down the cost of formal methods so that we get something that looks like the third bar.

Figure 1.1: Cost of software development with formal verification vs. without

structure invariants. VST has been around for a few years and is starting to be used for many practical verifications. The system was used recently to verify an HMAC algorithm [16]. This verification took several months and was quite tedious.

The goal of this thesis is to develop techniques that make it more practical to use an interactive theorem prover to verify complex data structure invariants. Our development efforts are optimized towards building a practical tool. This means that some proofs and tactics will not be fully verified with respect to Coq's primitives.

1.1 Specifications

The first question in writing a specification is what needs to be specified and in what language. Some researchers have attempted to completely specify a program's behavior, with the aim of is fully verifying the software to assure there are absolutely no bugs. Unfortunately, these specifications are often more complicated than the software itself, and the veracity of the specification itself becomes a significant source of potential error.

An alternative approach is to instead focus on verification which will more fruitfully point out bugs in the program. One such method is to specify that a program does not have certain ill-formed behaviors such as out of bound array references and memory leaks. In order to verify these properties, one will need to build up data structure invariants. Verifying these invariants will likely reveal many important bugs; this is our focus.

1.2 Proof development productivity

A primary focus of this thesis is the improvement of proof development productivity. Interactive theorem provers are quite tedious. Every man hour of development requires over 100 man hours of verification time, and this ratio needs to be significantly improved before theorem proving becomes viable for software verification.

We divide productivity issues into two categories. The first is that of dealing with

the performance of the theorem proving tool itself – theorem provers can be vastly slower than compilers and other traditional development tools. The second issue is that workflows are different for theorem provers than for software development: formal theorems and proofs are much more brittle to change. As an example, proving a theorem may reveal errors in the statement of the theorem itself. After the theorem statement is corrected, one needs to replay and update the potetially long and complex proof script.

1.2.1 Challenges of using the Coq theorem prover

We have chosen to use the Coq theorem prover [58] for verification. While Coq is a sophisticated theorem prover, when scaling up to complex theorems the tool slows down substantially. In order to maintain good performance, both the length of proof scripts and the statement of the theorems need to be kept small. But, dividing theorems into small chunks is a non-trivial task. There is a second issue in that many tactics (programs which automatically assist in constructing proofs) do not transform goals in exactly the right way to complete a theorem. This lead us to use a deep model in which we use data structures to represent the program invariants. This allows us to create tactics that better model the types of transformations that we need.

We have developed our own PEDANTIC separation logic framework which is similar to VST and Iris. The logic is simplified compared to these other tools, with a focus more on data structure invariant specification. Unlike VST or Iris, PEDANTIC

does not have concurrency primitives. Many important bugs do not involve concurrency. Having a usable tool even without concurrency reasoning would be extremely valuable. PEDANTIC, unlike VST or Iris, uses a deep model to allow greater flexibility in constructing tactics. At the time we started this work, Iris did not exist and VST was still immature. At this point, if starting over VST would be a better choice than developing our own logic. However, VST is running into many of the same issues that we are seeing with proof development in PEDANTIC.

Even with PEDANTIC's customized tactics, verifying a 200-line program with complex invariants is extremely tedious. Often it is not possible to break up a theorem into small pieces and the Coq theorem prover can become quite slow and may run out of memory. As part of our research, in addition to working towards the DPLL verification, we also developed two tools to improve proof development productivity.

First, we developed a new IDE, CoqPIE, specifically aimed at improving productivity of doing large proof development tasks using Coq. The tool introduces several innovations. First, all intermediate Coq goal states are cached. This eliminates the need to rerun Coq when moving back and forth in a proof. Second, a replay mechanism has been developed to assist in updating proofs when a lemma changes. Third, we developed a project browser that allows one to quickly navigate a large proof development with several source files. There are many IDE efforts for Coq; however, none hit on all of the important productivity issues. For example, PIDE [78] introduces a mechanism for rerunning proof scripts concurrently, which improves feedback. How-

ever, one can get similar results by having the IDE replace proof scripts with `admit` in all lemmas not being worked on by the user. PIDE also does not support replaying proof scripts.

In addition to CoqPIE, we developed our own advanced rewriting library to simplify Coq expressions. Our rewriter integrates many different algebraic simplification algorithms. To ensure termination, recursive path orderings are used [34]. We also use conditional rewrite rules and automate the necessary matching and reasoning for associative/commutative operators. We also include rules to compute transitive closures of equalities and inequalities, and to incorporate contextual rewriting. An expression $a \rightarrow b$ means the system can assume a is true when simplifying b; this leads to many important simplifications of the sort not currently possible with the existing Coq simplifiers. The rewrite library gives Coq a simplification mechanism that competitive with that of the Isabelle rewriter. [62]. The rewrite library also serves as a general purpose theorem proving tactic. If a goal rewrites to `True`, then the theorem is proven. This turns out to be useful as there are many intuitively obvious goals that are difficult to prove using Coq's primitives.

Also, in the process of developing or rewriting library we found many optimizations that could substantially improve Coq's performance. For example, Coq uses a unary representation for integers where an integer is either 0 or the successor of another integer. The number 500 takes a linked list of 500 records to represent and in fact Coq cannot handle integers larger than about 30,000 (note this will eventually be

fixed in the upcoming 8.8 release of Coq).

1.3 The DPLL algorithm verification

As a case study to stress test the tools needed for a practical interactive theorem proving, we have chosen to verify DPLL [77], a SAT solving algorithm. For this project we are using a simplified implementation which is around 200 lines of C code and which does not employ learning. This algorithm still contains many complex data structures and invariants that exemplifies what exists in many real software systems. The source code can be found in Appendix D.

DPLL is a SAT solving algorithm which finds assignments for the boolean variables of a sentential logic expression in conjunctive normal form. One of the key features of this algorithm is a watch variable algorithm that marks two unassigned variables in each clause. When any clause gets to a point that all variables are assigned except one and none of the variables have an assignment that makes the clause true, then we can deduce the assignment of the last variable. The rules for choosing (and moving around) watch variables are relatively complex and provide a good exercise for analyzing verification techniques.

1.4 Contributions of the work

The most important contribution of this thesis is to gain a greater understanding of the issues in verifying complex algorithms. The DPLL verification gives a good introduction to the types of proofs that need to be completed.

While the DPLL verification is not complete, many key pieces have been finished. In the process of verifying these pieces we found the need for significant improvements in prover productivity, which led us into working on that task: the CoqPIE UI introduces many features important for large proof development, and our rewriting library introduces a much more powerful rewriting simplifier for Coq.

1.5 Organization of thesis

Chapter 2 outlines our PEDANTIC framework. It is a set of definitions and data types which defines a language for representing separation logic based state assertions, a language for imperative programs and tactics to reason about Hoare triples using these languages. Since theorem proving is tedious, we developed an IDE to improve productivity when working with large theorems. Our CoqPIE IDE is described in chapter 3. One of the key features of our system is the ability to simplify invariants. Part of the framework includes tactics to rewrite separation logic expressions. This tactic turned out to be quite slow and takes up 80% of the runtime when running proof scripts for our DPLL verification. We describe an ML-plugin that we developed

for Coq in chapter 4 which is considerably faster. The tactic provides a powerful general purpose framework for rewriting. This chapter also discusses how this tactic plugin will eventually be integrated into PEDANTIC. Finally we describe the work that has been done towards verifying DPLL using PEDANTIC in chapter 5.

Chapter 2

The PEDANTIC framework

Many software bugs can be traced to complex data structure invariant violations. As an example, the DPLL algorithm we are verifying uses 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. Correct program function requires many different correlations to hold between these two data structures. More details on the DPLL verification are in Chapter 5. In this Chapter we define the separation logic framework in which programs such as DPLL can be verified. 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 [14, 26, 54]. The framework implements a logic language based on separation

logic [72] 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. CoqPIE is described in chapter 3. This chapter 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 in this chapter as well as the DPLL verification described in Chapter 5) 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 produces 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.

2.1 Overview of PEDANTIC contributions

In order to create an easy to use yet extensible framework, we took the work of [8, 14, 26, 54, 59] 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.

2.1.1 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. Gallina is the functional programming language which is the foundation of Coq's logic. This gives us greater flexibility in designing tactics. It also separates the design of the algorithm from the verification of the tactic. Figure 2.3 shows the Coq declarations for the data type representing separation states. Once a correctness theorem for a tactic is created in the framework, it never has to be

```
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) {
ℓ2.1.1   struct list *i = NULL, *n, *x;
ℓ2.1.2   struct tree *t = r;
ℓ2.1.3   p = NULL;
ℓ2.1.4   while (t) {
ℓ2.1.5    n = p;
ℓ2.1.6    p = malloc(sizeof(struct list));
ℓ2.1.7    p->t = t;
ℓ2.1.8    p->n = n;
ℓ2.1.9    if (t->l==NULL && t->r==NULL) {
ℓ2.1.10     if (i==NULL) {
ℓ2.1.11      t = NULL;
ℓ2.1.12     } else {
ℓ2.1.13      struct list *tmp = i->n;
ℓ2.1.14      t = i->t;
ℓ2.1.15      free(l);
ℓ2.1.16      i = tmp;
ℓ2.1.17     }
ℓ2.1.18    } else if (t->r==NULL) {
ℓ2.1.19     t = t->l;
ℓ2.1.20    } else if (t->l==NULL) {
ℓ2.1.21     t = t->r;
ℓ2.1.22    } else {
ℓ2.1.23     n = i;
ℓ2.1.24     i = malloc(
                sizeof(struct list));
ℓ2.1.25     i->n = n;
ℓ2.1.26     x = t->r;
ℓ2.1.27     i->t = x;
ℓ2.1.28     t = t->l;
ℓ2.1.29    }
ℓ2.1.30   }
ℓ2.1.31  }
```

Figure 2.1: Example program which builds a linked list of all nodes in a tree

regenerated when verifying a program. As an example of how Gallina functions work, we defined a function to simplify states defined with the `absState` data structure as follows:

```
Fixpoint simplifyState (context : list Context) (s : absState)
         : absState :=
     simplifySt (fun c e => (basicExpRule1 c e)) srule context s.
```

We later use this definition in many lemmas that helps us with program verifications. An example is as follows:

```
Theorem simplifyEquivb1 : forall (s : absState) s' b state,
    s' = simplifyState nil s ->
    realizeState s b state ->
    realizeState s' b state.
Proof.
...
```

2.1.2 Embedding pointers

The Btree example in [54] 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.

2.1.3 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. This pairing process has the advantage that it avoids the need for the user to reenter major portions of the resulting state at merge points and for implications, it automates portions of the entailment verification.

2.1.4 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.

2.1.5 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 ex-

tensibility. 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 Gallina 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.

2.2 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 2.1 shows a C-like program which performs a traversal of a tree, given in the parameter `r` to the function `build_pre_order` and places the result in a linked list `p`. This function contains a pointer `t` which walks the tree. As the tree is walked, elements are added to the `p` list.

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

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

2.2.1 PEDANTIC representation of programs

As a first step in verifying our program, we translate the C code to an imperative langauge defined by Coq data types. Programs are currently translated by hand which is a potential source of errors; eventually, a parser should be developed to perform the translation. Figure 2.2 shows the Coq data structure used to represent programs and expressions. It is based on Pierce's Software Foundations library [65], with constructs added for managing heap data structures. We have also added function calls/return and constructs for catch/throw commands. However, we have not used them in our programs as yet. Appendix A shows our translation of the C program to the PEDANTIC representation for Coq.

The `aexp` structure in figure 2.2 represents an expression. `ANum` is a constant value, `AVar` is a variable `APlus`, `AMinus`, `AMult` are arithmetic operators. `AEq` is equality. `ALe` is less than or equal. `ALand`, `ALor` and `ALnot` are boolean operators.

Programs are built from the `com` type. We have the control operators `CIf` and `CWhile` for if and while loops. We have `CCall` and `CReturn` for function calls and returns (not used in our current derivations). We have `CThrow` and `CCatch` for catch and throw (also not used). `CAss` is our assignment operator. It evaluates the expression and assigns the value to a variable. Note this operator cannot access the heap. We have `CLoad` which moves a value from the heap to a variable and `CStore` which evaluates an expression and saves the result in a location specified by evaluating the second parameter. `CNew` allocates space on the heap. It specifies a variable in which

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AVar : id -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp
  | AEq : aexp -> aexp -> aexp
  | ALe : aexp -> aexp -> aexp
  | ALand : aexp -> aexp -> aexp
  | ALor : aexp -> aexp -> aexp
  | ALnot : aexp -> aexp.

Inductive com : Type :=
  | CSkip : com
  | CLoad : id -> aexp -> com
  | CStore : aexp -> aexp -> com
  | CAss : id -> aexp -> com
  | CNew : id -> aexp -> com
  | CDelete: aexp -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : aexp -> com -> com -> com
  | CWhile : aexp -> com -> com
  | CCall : id -> id -> (list aexp) -> com
  | CReturn : aexp -> com
  | CThrow : id -> aexp -> com
  | CCatch : id -> id -> com -> com -> com.
```

Figure 2.2: PEDANTIC representation of imperative programs

to save the location of the start of the allocated cells as well as an expression which specifies how many cells to allocate. `CDelete` removes a block from the heap. It takes two parameters, an expression to evaluate representing the start of the block to delete and an expression to evaluate representing the size. More details on the semantics are discussed in section 2.4.

2.2.2 PEDANTIC Concrete States

Concrete states in PEDANTIC are represented by a pair (e, h) where e is a mapping from variable names to natural numbers. Variables are given a default value of 0. h is a partial mapping from natural numbers to natural numbers. It represents the information in the heap. The set $dom(h)$ represents the set of allocated memory cells. for any $l \in dom(h)$, $h(l)$ represents the value at that location.

2.2.3 Separation Invariants

In this section we describe the internal structure of PEDANTIC invariants in more detail. Invariants are assertions that are true for a subset of the concrete states represented by the pair (e, h) described in section 2.2.2. Figure 2.3 shows the Coq definitions used for our state assertions. We use the abbreviation $a * b$ for `AbsCompose` a b. In addition to the separation `*` operator, we also support the separation logic magic wand `AbsMagicWand a b` or `a` $-*$`b` which means subtract

```
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
 ...
```

Figure 2.3: Separation expression and assertion data structures

the data structures in `b` out of `a`. Magic wand can be viewed as an inverse of the * operator as $(a * b) - *b = a$. 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 [26] 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

$$
\begin{array}{ll}
(e,h) \vdash \texttt{AbsCompose}\ s_1 s_2 & \text{iff}\ \ \exists h_1 h_2, dom(h_1) \cap dom(h_2) = \emptyset \wedge \\
 & \quad (e,h_1) \vdash s_1 \wedge (e,h_2) \vdash s_2 \wedge h = h_1 \cup h_2 \\
(e,h) \vdash \texttt{AbsMagicWand}\ s_1 s_2 & \text{iff}\ \ \exists h_1 h_2, dom(h) \cap dom(h_2) = \emptyset \wedge \\
 & \quad (e,h_1) \vdash s_1 \wedge (e,h_2) \vdash s_2 \wedge h_1 = h \cup h_2 \\
(e,h) \vdash \texttt{AbsEmpty} & \text{iff}\ \ dom(h) = \emptyset \\
(e,h) \vdash \texttt{AbsExists}\ v.s & \text{iff}\ \ \exists x \in V.(e,h) \vdash s[x/v] \\
(e,h) \vdash \texttt{AbsAll}\ v \in R(\bar{p}).s & \text{iff}\ \ \forall x \in R(R(\ll \bar{p} \gg e)) \\
(e,h) \vdash \texttt{AbsUpdateVar}\ i\ v\ s & \text{iff}\ \ (e[i] = \ll v \gg e \wedge \exists x \in V.(e[x/i],h) \vdash s \\
(e,h) \vdash \texttt{AbsUpdateWithLoc}\ i\ v\ s & \text{iff}\ \ (e[i] = h(\ll v \gg e) \wedge \exists x \in V.(e[x/i],h) \vdash s \\
(e,h) \vdash \texttt{AbsUpdateLoc}\ l\ v\ s & \text{iff}\ \ (\ll v \gg e = h(\ll l \gg e) \wedge \\
 & \quad \exists x \in V.(e, h[x/ \ll l \gg e]) \vdash s \\
(e,h) \vdash \texttt{AbsUpdState}\ s_1\ s_2\ s_3 & \text{iff}\ \ (e,h) \vdash \texttt{AbsCompose}\ (\texttt{AbMagicWand}\ s_1\ s_2)\ s_3 \\
(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 \\
 & \quad \forall x \in 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 \\
\multicolumn{2}{l}{\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} \\
\multicolumn{2}{l}{(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} \\
\multicolumn{2}{l}{(e,h_n) \vdash \texttt{TREE}(h(\ll r \gg +f_0), v_{f_0}, s, \bar{f})\ \wedge \ldots \wedge} \\
\multicolumn{2}{l}{(e,h_{n+m-1}) \vdash \texttt{TREE}(h(\ll r \gg +f_{m-1}), v_{f_{m-1}}, s, \bar{f}) \wedge} \\
\multicolumn{2}{l}{\texttt{combine}(h,h_0,\ldots,h_{n+m-1}) \wedge} \\
\multicolumn{2}{l}{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} \\
\multicolumn{2}{l}{\text{where } \texttt{combine}(h,h_0,\ldots,h_n) \text{ iff } \forall i,j \leq n.dom(h_i) \cap dom(h_j) = \emptyset\ \ \wedge} \\
\multicolumn{2}{l}{\forall i \in dom(h). \exists j.i \in dom(h_j) \wedge h(i) = h_j(i)}
\end{array}
$$

Figure 2.4: Semantics for the separation predicates

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 constructs, `AbsPredicate`, `AbsCell` and `AbsTree`. `AbsLeaf AbsPredicate` (P :: `nil`), 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

`AbsExp` core cases				
$\ll c \gg e\ b$	$=$	c		constant
$\ll v \gg e\ b$	$=$	$e[v]$		environment variable
$\ll v \gg e\ b$	$=$	$b[v]$		quantifier variable
functions embedded in `absFun`				
$\ll l_1 + l_2 \gg e\ b$	$=$	`NatValue` $v_1 + v_2$	iff	`NatValue`$v_1 = \ll l_1 \gg e\ b \wedge$ `NatValue` $v_2 = \ll l_2 \gg e\ b$
		`NoValue`		otherwise
$\ll l_1 * l_2 \gg e\ b$	$=$	`NatValue` $v_1 * v_2$	iff	`NatValue` $v_1 = \ll l_1 \gg e\ b \wedge$ `NatValue` $v_2 = \ll l_2 \gg e\ b$
		`NoValue`		otherwise
$\ll l_1 = l_2 \gg e\ b$	$=$	`NatValue` 1	iff	$\ll l_1 \gg e\ b = \ll l_2 \gg e\ b$
		`NatValue` 0		otherwise
$\ll l_1 < l_2 \gg e\ b$	$=$	`NatValue` 1	iff	`NatValue` $v_1 = \ll l_1 \gg e\ b \wedge$ `NatValue` $v_2 = \ll l_2 \gg e\ b \wedge$ $v_1 < v_2$
		`NoValue`		otherwise
		`NatValue` 0	iff	`NatValue` $v_1 = \ll l_1 \gg e\ b \wedge$ `NatValue` $v_2 = \ll l_2 \gg e\ b \wedge$ $v_1 \not< v_2$
		`NoValue`		otherwise
...				
$\ll$ `find`$(l, v) \gg$ e b	$=$	$F(\ll l \gg e\ b, \ll v \gg e\ b)$		
where		$F(l, v) = v$	iff	$v =$ `ListValue`$[l, \ldots]$
		$F(l, v) = v_i$	iff	$v =$ `ListValue`$[l', v_1, \ldots, v_n] \wedge$ $v_i =$ `ListValue`$[l, \ldots] \wedge$ $i \in [1, \ldots, n]$
		$F(l, v) =$ `NoValue`		otherwise
$\ll$ `rangeSet`$(f) \gg$ e b	$=$	$RS(\ll f \gg e\ b)$		
where		$RS(v) = l + +$ $RS(v_1) + +$ $\ldots + + RS(v_n)$	iff	$v =$ `ListValue`$[l, \ldots]$
		$RS(v) = []$		otherwise
$\ll$ `rangeNumeric` $(s, f) \gg$ e b	$=$	$RN(st, en)$	iff	$\ll s \gg e =$ `NatValue` $st \wedge \ll f \gg e\ b =$ `NatValue` $en \wedge$
where		$RN(st, en) = [st, \ldots, en]$		

Figure 2.5: Semantics for expressions

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 2.4. In the figure, (e, h) represents a concrete state. $e \in \texttt{Id} \to \texttt{Nat}$ represents assignments for program variables and $h = \texttt{Nat} \to \texttt{option Nat}$ represents the values on the heap. $(e, h) \vdash s$ means that (e, h) satisfies the assertion s. The notation $\ll exp \gg e$ represents evaluation of the expression *exp* with the set of variables in the environment e.

2.2.4 Update predicates

In addition to the basic constructs of separation logic, we have introduced a number of "update predicates", `AbsUpdateVar`, `AbsUpdateLoc`, `AbsUpdateWithLoc` and `AbsUpdateState`. These predicates allow us to define rules that make forward propagation over statements automatic. As an example, if we have the Hoare triple { `p` }`v := e`{`?`}, our assignment rule from figure 2.7 will fill in the `?` with `AbsUpdateVar p v e`. It will later be up to the simplification engine to remove these update predicates from the state. The `UpdateWithLoc` predicate is used to forward propagation over statements of the form `CLoad loc var` which loads a value from the heap into a variable. The `UpdateLoc` constructor is used to propagate over statements of the form `CStore value loc` where a value is being saved into the heap. Finally `AbsUpdateState s m p` is an abbreviation for $(s - *m) * p$.

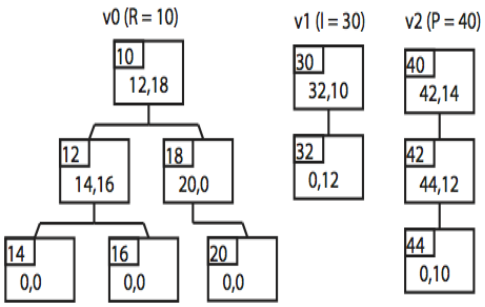
2.2.5 Expressions and values

In the PEDANTIC imperative language semantics, values are exclusively naturals. The expressions of figure 2.2 only have operators over naturals. Boolean values are also represented by naturals following the C convention (0 for false and anything else for true).

The expressions in the abstract state language shown in figure 2.3 need values that can either be naturals or more complex pieces of data. To satisfy this constraint, the values for our separation 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.
     ...
```

In the section 2.2.6 we will need these more complex values to create functional representations of list and tree data structures. Separation logic 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 2.5. A more detailed explanation of the latter is given in the next section where we show a sample assertion.



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

Figure 2.6: A possible program heap state and PEDANTIC assertion

2.2.6 An example state assertion

To better understand state assertions we present a sample heap snapshot of our example program in Figure 2.6. 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 [54]. 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

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

$$\frac{canAccess(e, s)}{\{s\}\ \texttt{CLoad x e}\ \{\texttt{AbsUpdateWithLoc}\ x\ e\ s\}}\ \text{load}$$
where $canAccess(l, s) = \forall eh.(e, h) \vdash s \rightarrow h(l) \neq \texttt{None}$

$$\frac{canAccess(x, s)}{\{s\}\ \texttt{CStore x e}\ \{\texttt{AbsUpdateLoc}\ s\ l\ x\}}\ \text{store}$$

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

$$\frac{\texttt{n} : nat}{\{s\}\ \texttt{v} := \texttt{NEW n}\ \{\exists x.s[x/\texttt{v}] * \texttt{v} \mapsto c_0 * \ldots * \texttt{v} + \texttt{n} - 1 \mapsto c_{\texttt{n}-1}\}}\ \text{new}$$

Figure 2.7: Forward propagation rules

`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 $[\texttt{10}, \texttt{12}, \texttt{14}, \texttt{16}, \texttt{18}, \texttt{20}]$. 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, $\texttt{find}([\texttt{30}, [\texttt{32}, [\texttt{0}], \texttt{12}], \texttt{10}], \texttt{32})$ will return $[\texttt{32}, [\texttt{0}], \texttt{12}]$. Finally, the last line states that either `T` is `0` or that `T` points to an element in `R`.

2.3 Verifying the initialization code

PEDANTIC uses Hoare triples to represent program verification goals. As an example, the initialization code for our tree traversal code creates an initial state in

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

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

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

$$\frac{\texttt{asUpdateWithLoc}\ v\ e\ s}{s'}\ \text{absUpdateWithLoc elim3}$$

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.

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

Figure 2.8: Rules for eliminating update constructs

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

Figure 2.9: Rule for eliminating magic wand

which our loop invariant holds. At the beginning of this code, it is only known that the variable `R` is the root of a tree on the heap and that the heap does not contain any other data. At the end of the code, we need to verify that the loop invariant holds. This is represented in Goal 2.3.1.

Goal 2.3.1

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

The assertion before the statements is what we know to be true before execute. The statement afterwards is what we want verified. As a first step, we forward propagate the precondition as shown in Goal 2.3.2. The `assignment` rule from Figure 2.7 is applied three times resulting in the state assertion at the bottom which contains three `AbsUpdateVar` constructs. The formal definition of `AbsUpdateVar` is given in Figure 2.4. Basically, it is an assertion that the state is exactly the state specified by its third parameter except that the variable specified in the first parameter is given the value of the second parameter. We also note that after propagating over the third statement, the assertion does not look like the postcondition that we want.

Goal 2.3.2

$\{$ AbsExists v_0.TREE$(R, v_0, 2, [0,1])$ $\}$
T := R;
$\{$ AbsExists v_0.AbsUpdateVar T R (TREE$(R, v_0, 2, [0,1])$) $\}$
I := 0;
$\{$ AbsExists v_0.AbsUpdateVar I 0
(AbsUpdateVar T R (TREE$(R, v_0, 2, [0,1])$) $\}$
P := 0;
$\{$ AbsExists v_0.AbsUpdateVar P 0
(AbsUpdateVar I 0
(AbsUpdateVar T R (TREE$(R, v_0, 2, [0,1])$)) $\}$

Our next step is to verify that this final assertion entails the desired post condition. This is the goal shown in Goal 2.3.3. We first need to simplify that final post-condition. This gives us the antecedent at $\ell_{2.3.3.1}$ in Goal 2.3.3.

Goal 2.3.3

$\ell_{2.3.3.1}$ { AbsExists v_0.
$\ell_{2.3.3.2}$ TREE$(R, v_0, 2, [0,1])*$
$\ell_{2.3.3.3}$ $[R = T]*$
$\ell_{2.3.3.4}$ $[I = 0]*$
$\ell_{2.3.3.5}$ $[P = 0]$ }

$\longrightarrow$

{ AbsExists v_0. AbsExists v_1. AbsExists v_2.
$\ell_{2.3.3.6}$ TREE$(R, v_0, 2, [0,1])*$
$\ell_{2.3.3.7}$ TREE$(I, v_1, 2, [0])*$
$\ell_{2.3.3.8}$ TREE$(P, v_2, 2, [0])*$
$\ell_{2.3.3.9}$ AbsAll $v_3 \in$TreeRecords(v_1).
$[$nth(find$(v_1, v_3), 2)$ inTree $v_0]*$
$\ell_{2.3.3.10}$ AbsAll $v_3 \in$TreeRecords(v_2).
$[$nth(find$(v_2, v_3), 2)$ inTree $v_0]*$
$\ell_{2.3.3.11}$ $[T = 0 \vee T$ inTree $v_0]$ }

Now we need to verify the entailment. Much is done by pairing off corresponding pieces in the antecedent and consequent. The verification proceeds in several steps:

1. The TREE construct at $\ell_{2.3.3.2}$ is paired with the TREE construct at $\ell_{2.3.3.6}$.

2. The predicate at $\ell_{2.3.3.4}$ is paired with the TREE at $\ell_{2.3.3.7}$. The null root variable is an empty tree.

3. A similar logic is used to pair the predicate at $\ell_{2.3.3.6}$ with the TREE at $\ell_{2.3.3.8}$.

4. Next, the predicate at $\ell_{2.3.3.4}$ and TREE construct at $\ell_{2.3.3.7}$ are used to deduce that the TreeRecords function at $\ell_{2.3.3.9}$ returns an empty set making the AbsAll trivially true.

5. Similar logic using the predicate at $\ell_{2.3.3.5}$ and TREE construct at $\ell_{2.3.3.8}$ are

used to deduce that the `TreeRecords` at $\ell_{2.3.3.10}$ function returns an empty set making the `AbsAll` trivially true.

6. Finally, the predicate at $\ell_{2.3.3.3}$ is used to infer that `T` is the root of the `TREE` represented by v_0 and hence in the tree. This verifies the right hand side of the disjunct at $\ell_{2.3.3.11}$

These steps verify that all pieces of the consequent and our original verification goal for the initialization code is complete.

2.4 Verifying the main loop

We now show how to verify that the invariant of figure 2.6 holds throughout our program. Appendix B shows the top level Coq proof that the invariant holds throughout the main loop of the program shown in Appendix A and ultimately the program in figure 2.1. 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 2.6.

This main proof generates 12 lemmas that need to be verified. `treeRef1` and `treeRef2` verify that a heap location being read belongs to an allocated block. `storeCheck1` and `storeCheck2` similarly verify that a heap location being writ-

ten 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. The next three subsections describe portions of the proof in more detail.

2.4.1 Forward propagation

Figure 2.7 shows the rules for forward propagating over a statement. 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[1]. To show how this propagation works, below is shown the state before (State 2.4.1) the assignment `N ::=!P` in Appendix B at ℓ_{B1}:

[1]Note that in our presentation, we switched the order of the parameters. The first parameter in the actual code is the state expression, the second and third are the location being updated and its location. This was done to make the PEDANTIC state assertions easier to read.

State 2.4.1

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

After the assignment, we get the state shown below with changes in blue (State 2.4.2). The rule `assignment` from Figure 2.7 is used to do this propagation.

State 2.4.2

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

We can then propagate over the next statement, $\mathtt{NEW\ P},\ \mathtt{ANum}(\mathtt{Size\_l})$ at ℓ_{B2} and we get State 2.4.3 as shown below. The rule `new` from figure 2.7 is used for this step.

State 2.4.3

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

2.4.2 Merging states

Shown below (States 2.4.4 and 2.4.5) are the two states from the ends of then- and else- blocks of the if statement when `mergeTheorem1` of Appendix B (at ℓ_{B3}) 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. The blue highlighting show pieces that will be simplified in the next step.

State 2.4.4

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

State 2.4.5

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

We start by doing simplifications and eliminations. The results are shown below in States 2.4.6 and 2.4.7 which represent simplifications of the States 2.4.4 and 2.4.5 respectively. Rules for eliminating `updateVar`, `updateLoc` and `updateWithLoc` are shown in figure 2.8. 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.

State 2.4.6

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

State 2.4.7

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

On the right hand side, we need remove the magic wand . To do this, we first need to unfold the tree rooted at v_5. The result of unfolding is shown below in State 2.4.8.

State 2.4.8

$$
\begin{array}{l}
(\texttt{AbsExists}\ v_8\ (\texttt{AbsExists}\ v_7 \\
\quad(\texttt{AbsExists}\ v_6\ (\texttt{AbsExists}\ v_5(\texttt{AbsExists}\ v_4 \\
\quad\quad(\texttt{AbsExists}\ v_3\ (\texttt{AbsExists}\ v_2(\texttt{AbsExists}\ v_1 \\
\quad\quad\quad(\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ v_6 \\
\quad\quad\quad\quad(\texttt{AbsUpdateWithLoc}\ \texttt{T}\ (v_6+1) \\
\quad\quad\quad\quad\quad(\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_r}\ (\texttt{T}+1) \\
\quad\quad\quad\quad\quad\quad(\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ \texttt{T} \\
\quad\quad\quad\quad\quad\quad\ (\texttt{P} + 1 \mapsto \texttt{T} * \texttt{P} \mapsto \texttt{N} * [\texttt{P} = v_4] * [0 < \texttt{T}]* \\
\quad\quad\quad\quad\quad\quad\quad\texttt{TREE}(\texttt{R}, v_1, 2, [0,1])* \\
\quad\quad\quad\quad\quad\quad\quad v_6 + 1 \mapsto v_8 * v_6 + 0 \mapsto nth(v_7, 0)* \\
\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(nth(v_7, 0), nth(v_6 :: v_7 :: v_8 :: nil, 1), \\
\quad\quad\quad\quad\quad\quad\quad\ 2, [0])* \\
\quad\quad\quad\quad\quad\quad\quad\texttt{TREE}(\texttt{N}, v_3, 2, [0])* \\
\quad\quad\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TreeRecords}(v_6 :: v_7 :: v_8 :: nil). \\
\quad\quad\quad\quad\quad\quad\quad\ [\texttt{nth}(\texttt{find}(v_6 :: v_7 :: v_8 :: nil, v_0), 2) \\
\quad\quad\quad\quad\quad\quad\quad\ \texttt{inTree}\ v_1]* \\
\quad\quad\quad\quad\quad\quad\quad\texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_3). \\
\quad\quad\quad\quad\quad\quad\quad\quad[\texttt{nth}(\texttt{find}(v_3, v_0), 2)\ \texttt{inTree}\ v_1]* \\
\quad\quad\quad\quad\quad\quad\quad[\texttt{T}\ \texttt{inTree}\ v_1]) * [\texttt{N} = v_5)]))* \\
\quad\quad\quad\quad\quad[\texttt{tmp\_l} = 0] * [\texttt{tmp\_r} = 0] * [0 > v_6]) * - \\
\quad\quad(\texttt{AbsExists}\ v_1(\texttt{AbsExists}\ v_0 \\
\quad\quad\ v_6 + 1 \mapsto v_1 * v_6 \mapsto v_0))* \\
\quad\quad[\texttt{I} = \texttt{tmp\_l}])))))))))
\end{array}
$$

We now use the rule shown in figure 2.9 to eliminate the magic wand construct. The result after simplification is State 2.4.9.

State 2.4.9

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

We next merge the two States 2.4.6 and 2.4.9. The result is shown below in State 2.4.10. Most of the work can be done automatically by pairing off identical components in the two branches. There is one term, $[\texttt{T} = 0 \vee \texttt{T}\ \texttt{inTree}\ v_1]$, that the user needs to introduce. It is derived from the fact that that $[\texttt{T}\ \texttt{inTree}\ v_1]$ is in the left state and $[\texttt{T} = 0]$ is in the right state. They can be put together in a disjunction.

One important subtlety of the pairing off is that decisions need to be made on how variables introduced by `AbsExists` are paired. For example, v_1 on the left in State 2.4.6 is paired with v_2 on the right in State 2.4.9. One should first note that all `AbsExists` quantifiers have been moved to the top level of the expression. The way this pairing is achieved is that when pairing off either two `TREE` constructs or two constructs of the form $l \mapsto v$ construct, we require an exact match (where appropriate substitutions are done for bound variables that have been mapped) of the first parameter of either construct. If the first parameter contains bound variables that have not been mapped, then the pairing is not allowed. Mappings can be introduced if the

remaining parameters involved in the mapping have bound variables in corresponding positions that need to be mapped together. Constructs other than `TREE` or $l \mapsto v$ must have all bound variables mapped and then match exactly (after bound variables are mapped).

Consider what happens when the term $\texttt{TREE}(I, v_2, 2, [0, 1])$ from State 2.4.6 is paired with $\texttt{TREE}(I, v_5, 2[0, 1])$ in State 2.4.9. This pairing is allowed since the first parameter of each `TREE` are identical. The pairing has the effect of introducing a mapping from v_2 in the former state to v_5 in the latter state. We also get a mapping from v_1 to v_2 by pairing $\texttt{TREE}(R, v_1, 0, [0, 1])$ in State 2.4.6 to $\texttt{TREE}(R, v_2, 0, [0, 1])$ in State 2.4.9. These two mappings allows

$$\texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_2).[\texttt{nth}(\texttt{find}(v_2, v_0), 2)\ \texttt{inTree}\ v_1]*$$

from State 2.4.6 to be paired to

$$\texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_5).[\texttt{nth}(\texttt{find}(v_5, v_0), 2)\ \texttt{inTree}\ v_2]*$$

in State 2.4.9. The result is shown in State 2.4.10.

State 2.4.10

(AbsExists v_6 (AbsExists v_5 (AbsExists v_4
(AbsExists v_3 (AbsExists v_2 (AbsExists v_1
$[\mathtt{I} = \mathtt{I}] * (\mathtt{P}+1) \mapsto v_6 * \mathtt{P} \mapsto \mathtt{N} * [0 < v_5] *$
$\mathtt{TREE}(\mathtt{R}, v_1, 2, [0,1]) *$
$\mathtt{TREE}(\mathtt{I}, v_2, 2, [0]) *$
$\mathtt{TREE}(\mathtt{N}, v_3, 2, [0]) *$
$\mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2).$
$[\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_1] *$
$\mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_3).$
$[\mathtt{nth}(\mathtt{find}(v_3, v_0), 2)\ \mathtt{inTree}\ v_1] *$
$[v_6\ \mathtt{inTree}\ v_1] *$
$[\mathtt{T} = 0 \vee \mathtt{T}\ \mathtt{inTree}\ v_1])))))))$

As a final step, we want to fold back the tree rooted at N with the cells at locations P and P + 1. The blue text in State 2.4.10 indicates components that will be folded. The result then entails the state that is shown below in State 2.4.11 (blue text are folded or newly added terms).

State 2.4.11

(AbsExists v_3 (AbsExists v_2 (AbsExists v_4
$\mathtt{TREE}(\mathtt{P}, v_3, 2, [0]) *$
$[\mathtt{nth}(\mathtt{nth}(\mathtt{v_3}, 1), 0) = \mathtt{N}] * [0 < \mathtt{nth}(\mathtt{v_3}, 0)] *$
$\mathtt{TREE}(\mathtt{R}, v_1, 2, [0,1]) *$
$\mathtt{TREE}(\mathtt{I}, v_2, 2, [0]) *$
$\mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_2).$
$[\mathtt{nth}(\mathtt{find}(v_2, v_0), 2)\ \mathtt{inTree}\ v_1] *$
$\mathtt{AbsAll}\ v_0 \in \mathtt{TreeRecords}(v_3).$
$[\mathtt{nth}(\mathtt{find}(v_3, v_0), 2)\ \mathtt{inTree}\ v_1] *$
$[\mathtt{T} = 0 \vee \mathtt{T}\ \mathtt{inTree}\ v_1])))$

2.4.3 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.

2.5 Related Work

Separation logic was originally developed by John Reynolds [72]. Tools based on separation logic were developed and found to be effective in finding bugs in Microsoft device drivers [40, 15, 64].

The basic concepts were later introduced in the Coq theorem prover. McCreight developed a set of axioms and tactics to reason about separation logic [59]. Systems such as Charge! [14], Bedrock [27], VST [8, 9] and IRIS [49] 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 [16]. 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 [50] and hence the C language semantics in VST are fully verified.

2.6 Conclusion

This chapter has covered the basic concepts of the PEDANTIC framework, a tool for verifying the correctness of imperative 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 2.6 is complete but correctness proofs of most auxiliary Lemmas still need to be completed and are for now `admit`ted.

We are also working on the verification of the DPLL [77] based SAT solver invariant. Work on this verification is described in Chapter 5. 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 described in Chapter 3. 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 entailment proofs and merges.

The biggest challenge in making interactive theorem proving tools practical for software development is to better understand proof development productivity issues. The tree traversal verification presented in this Chapter consists of about 1100 lines of Coq code. It took the author a few weeks to develop the verification though this included work debugging the PEDANTIC infrastructure. It likely would have taken 1-5 days if the PEDANTIC infrastructure didn't need fixing. Productivity becomes considerably worse for larger programs as wee will see in Chapter 5. 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.

Chapter 3

The CoqPIE IDE

Large-scale proof development in Coq can be tedious. Developing a proof script may reveal errors in the statement of the theorem, and correcting these errors often requires proof scripts to be updated. Original goal information is often not available, making it harder to figure out what needs to change. Moving forward or backward in a proof can be very slow if there are large proof goals and many tactics. When moving from one step to a subgoal it may be that only a small portion of the hypotheses changed, but that change can be difficult to find if there are many complex hypotheses. These are just a few of the shortcomings of existing Coq workflow. The development of our DPLL verification has been quite tedious due to these issues as described in Chapter 5. These shortcomings have made proof development much slower than it should be.

In this chapter we present CoqPIE, a new development environment for Coq which

delivers editing functionality centered around common prover usage workflow that is not found in existing tools. The design is aimed at resolving the workflow issues mentioned above. In order to facilitate much of the functionality of the tool, a Coq parser is integrated into the editor. The parser generates an AST, and nodes in the AST are linked to positions in the source text. While the editor is not fully aware of the Coq semantics of terms, many properties of the declarations such as dependencies can be extracted.

This parsed AST is used to implement the most novel features of CoqPIE: the ability to extract lemmas and to replay/patch proof scripts automatically. None of the existing Coq IDE tools can exploit ASTs in this way. CoqPIE also includes other useful features to help streamline proof development. Support for pre-running an entire project and saving each proof step for later reference is integrated; this allows fast movement around the proof tree without the need to replay proof scripts. A tree view window is presented which summarizes all files and definitions in the project and aids rapid movement around a project. Difference highlighting using the parsed ASTs allows for quick identification of portions of a term that have changed. Dependencies are extracted from the ASTs and declarations impacted by edits are marked.

The need for a fancier IDE is arguably not uniform across all uses of Coq. If assertions are small and tactics are powerful, the existing tools can work. Additionally, uses of reflection [42, 53] and first order tactics [12] can produce smaller proof scripts. When the tactics are more intelligent, fewer direct references to hypotheses need to

be made and hence the proofs will be more robust to changes in the proof statements. That said, improving the editing workflow will help all users – it is too much to rely only on tactics and reflection to simplify the proving process. The features of CoqPIE will give users greater flexibility in choosing a workflow.

We have been using CoqPIE exclusively for our DPLL proof development (Chapter 5). We never open Proof General or CoqIDE. This validates that the tool is reasonably robust. We also find we are using most of the functionality.

Despite this success, CoqPIE does have a few weaknesses in its current form. An initial processing phase has to process the whole file before editing can commence, and an incomplete dependency analysis necessitates periodic full rebuilds. We will discuss these trade-offs below.

We first review some common pitfalls in Coq proof development workflow, and then give an overview of CoqPIE and its features that are designed to improve prover productivity.

3.1 Proof development workflow

There is currently a disconnect between how existing Coq prover tools work and how the user is thinking about achieving a proof of their theorem in Coq. CopPIE aims to bridge this gap. In this section we describe some of the common workflows that are not well-supported by existing IDEs.

- A complex Coq theorem nearly always requires many supporting lemmas. And, the statement of the theorem may itself be quite complex. For example, in our own ongoing DPLL SAT solver verification, the statement of the program invariant fundamentally requires about 100 lines of Coq code. The main correctness theorem which states that the invariant is preserved at the end of execution expands to several hundred lines of Coq code because this large invariant needs to be stepped through around 200 lines of C code.

- When developing a complex proof, a common workflow is to work top down rather than bottom up: when a lemma is needed, the statement of the lemma is entered, it is `admit`'ted for now, and work can continue on the main proof. The lemmas can be discharged later.

- Proving a lemma often reveals a small error in the *statement* of the lemma. Changing the statement then requires other proofs that were using the lemma to be revised as well (this assumes the user was proceeding in top-down proof mode). Changing the statement of any theorem requires its proof script to be replayed and for the parameters to many tactics to be updated. The system that generated hypothesis `H3` may generate `H5` for that same hypothesis as a revised proof statement may have more hypotheses. This replay process can thus require considerable manual effort to patch up proofs.

- As the size of a proof grows, Coq's performance can slow down dramatically.

The number of tactics to complete a proof can easily get into the hundreds, and it can take a minute or more to process each tactic. This slow performance is compounded by the constant need to replay and revise scripts when definitions and lemmas change. This performance issue can sometimes be resolved by breaking a large proof into several smaller pieces, but this means there are now more proof statements to revise when errors are found.

- There is sometimes a need to compare large terms to find the small portions that differ. In our separation logic framework, we found that it is often necessary to prove that an invariant that has been permuted by several forward propagations is equivalent to the original invariant. It is critical to be able to quickly identify the few lines (potentially out of hundreds) that have actually changed.

Improving the above and similar workflows is the primary goal of the design of CoqPIE, which we now describe.

3.2 An overview of CoqPIE

The diagram in Figure 3.1 shows the CoqPIE UI with a sample proof derivation open. There are three views shown. On the left is a tree view of the entire project similar to the tree view found in modern IDEs. The top-level of the tree view shows the files in the project; opening a file node displays a list of all the Coq declarations in that file. Opening a theorem declaration in turn shows the steps used to prove

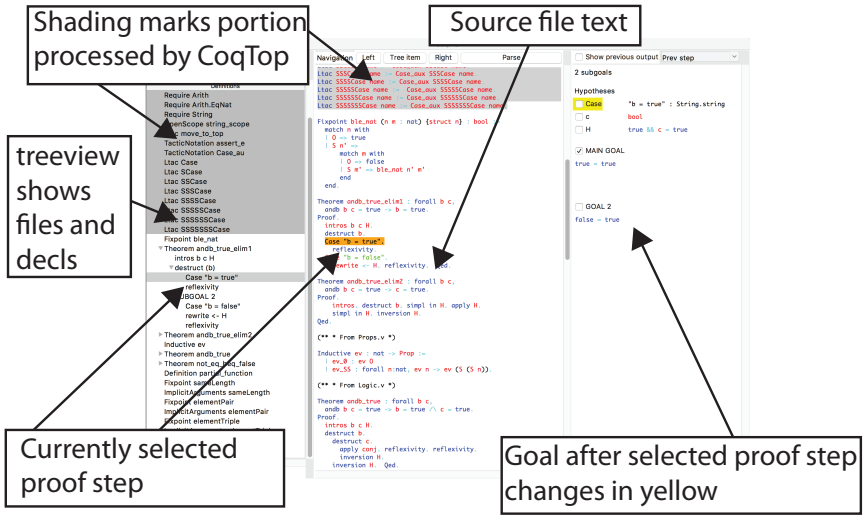


Figure 3.1: The main CoqPIE window

that theorem, with steps arranged in a tree based on subgoal relationships.

The middle view displays the source file based on the selection made in the tree view on the left. This view functions in a manner similar to the source file view in CoqIDE or Proof General. As with those tools, shading is used to indicate the portion of the file already processed to coqtop. Unlike Proof General and CoqIDE, the CoqPIE process management system automatically recompiles dependent source files.

The window on the right is similar to the Coq state window in CoqIDE or Proof General: it shows the current goal and hypotheses. However, unlike the other IDEs, instead of showing the state at the current processing point of Coq, it shows the state just *after* the selected definition or proof step from the tree view at the left. This is possible because CoqPIE runs the entire project and saves all output from Coqtop

before editing can commence. With this initial pass it is possible to very quickly browse theorems and to see the state after each step. This full proof tree state is also maintained during editing: as the user edits a source file and reruns coqtop to verify the updates, the cached outputs are updated. Differences from the state just before the most recent tactic was executed are highlighted in yellow. As an alternative, one can view differences between hypotheses and the goal or differences between old and new versions of a state (useful for the replay assist described later). A combobox just above the window on the right allows selection of which differences to show.

Since CoqPIE keeps intermediate proof state around it can be more intelligent about whether definitions and lemmas are up-to-date. Definitions with out-of-date Coq output information are color coded so the user knows they need to be replayed. CoqPIE can also do more intelligent prover state management using dependency information. When moving Coq processing to a specific point in a file, only known dependent files must be recompiled. More precisely, the script to build a CoqPIE project specifies a compilation order for the source files, and CoqPIE only needs to compile files before the one currently being edited. Furthermore, with calculated file dependency information only a subset of the earlier files need to be recompiled. File dependencies are easier to compute than definition dependencies as they are based on `Import` and `Export` statements. CoqPIE does not attempt to optimize recompilation inside a single file based on dependency relationships between definitions.

Additionally, when files are recompiled the scripts inside a proof are replaced

behind the scenes with `admit`. The only proof script compiled is the one for the theorem the user is actively working on. This gains the same performance advantage as concurrent processing in other systems such as that developed by [23]. The main difference is that CoqPIE does not launch background tasks to rerun the scripts inside the theorems. Instead, the user should run a background task of a complete project rebuild outside of CoqPIE to verify correctness of the derivation as a whole.

Unlike frameworks based on PIDE [2, 23, 82, 83], theorem proving in CoqPIE is only done when explicitly requested by the user. The CoqPIE paradigm is more like Proof General or CoqIDE in which the editor allows control of a REPL (Read Eval Print Loop) interface.

3.2.1 Parsing

Many of the important features of CoqPIE require that we implement a parser for Coq syntax. In addition to source files, we also need a parser for the proof state output that is generated after each tactic is applied. Nodes in the generated AST are linked to positions in the source text, which allows us to implement features such as incrementally expanding individual goals or hypotheses, and highlighting differences between terms. Additionally, macro operations such as extracting a lemma from the current goal state depend on having AST data available. We do not have a pretty printer for Coq definitions. Instead, our refactoring operations tend to copy text from the source file and then paste that text in new positions as required by the refactoring.

Hence, often, the original formatting of the text is preserved.

Coq has an internal AST data structure called CoqAst which is what we need, but it is not easily accessible with the current API. So, for the current implementation of CoqPIE, we chose to create our own parser. This choice has a number of ramifications. First, the Coq language is quite large and complex; we are only able to cover a subset. Second, Coq has a `Notation` construct that can add new syntax to the language. We currently do not have the capability to handle this construct. The longer-term strategy is to work with the Coq development team to use CoqAst and the same parser that is integrated into Coq.

Parsing is done incrementally during the editing of source files. CoqPIE keeps track of the locations at which each definition and tactic starts and ends, and only re-parses the modified definitions. This re-parsing takes a second or less for small definitions and a minute or two for large proof scripts (fifty lines or more) such as those in our DPLL verification.We mark in yellow the definitions that are affected by user edits; the user can then press the "Parse" button to accept the edits and update the parse tree. In the future we also plan to add a mechanism which will automatically reparse a small definition if the user has stopped typing for a few seconds.

3.2.1.1 Parse errors

Because our parser only covers a subset of Coq and also because users often enter erroneous inputs, it is important to handle parse errors gracefully. When a construct

cannot be parsed, we fall back on a more primitive parser. This parser simply looks for a "." (in a manner similar to Coqoon) to mark the end of a definition. For proofs, the system looks for the keywords `Proof` and `Qed` to mark the beginning and end of the proof section and then identifies tactics by looking for periods at the end.

3.2.1.2 Parsing goal states

The CoqPIE parser can parse the goal state output that is printed after a tactic is executed. This parsing may be slow if the state is complex. To address performance, goal states are only parsed when they are needed for display or for use by a refactoring operation. Once a state is parsed, the AST is cached so that it never needs to be parsed again. This is unfortunately still too slow the first time a large goal is parsed. When CoqPIE is integrated to use the internal Coq parser performance here should become reasonable.

3.2.2 Difference highlighting

Often it is useful to be able to highlight the differences between two complex terms. We have found three specific cases where this is useful. First, after applying a tactic, changes in the hypotheses can be highlighted. Second, one can highlight differences between a hypothesis and the goal to help in identifying transformations that need to be done to apply the hypothesis. Third, differences between definitions can be highlighted; in our DPLL example, we are using this to identify variations in

different versions of the program invariant. Since we are parsing the prover output we can compare ASTs to find differences, giving more accurate results than a raw textual comparison.

3.2.3 Dependency management

CoqPIE maintains dependencies between definitions and theorems. When a theorem or definition is changed, all dependent theorems and definitions are marked. We use color coding in the leftmost tree view window to indicate declarations that need recompiling. When a definition is edited, it is colored red. All definitions that depend on it are colored orange. There is similar coloring when the statement of a theorem is changed. However, when only the proof script is changed, then that theorem alone is colored yellow. Any dependent theorems are colored green.

Creating an exact algorithm for tracking dependencies is very difficult [7] due to the complexities of Coq's higher-order semantics. Also [68] discusses many issues in doing dependency analysis. They analyze opaque vs transparent proof dependencies. An opaque transparency is a dependency that can be identified by the proof statement alone. Transparent dependencies occur when a tactic in the proof script depends on another theorem. These can sometimes be hard to identify as theorems may be chosen automatically by tactics such as `auto`.

Our approach is to use an approximate dependency tracking algorithm: CoqPIE bases dependency relationships on identifiers that explicitly appear in a proof or defi-

nition. Transparent dependencies are not handled at this time. So, some dependencies will be missed; the repercussion of a missed dependency is that the user will not be alerted that a definition may need updating. The error will often appear when the user tries to recompile the definition in CoqPIE. If the error is inside a proof script, CoqPIE compilation may not catch the newly introduced error. In this case the error will not show up until the user tries to recompile the entire project outside of CoqPIE or edits that proof.

We made the decision in CoqPIE to avoid an internal verification of the full project to reduce overhead. The primary goal is to have a dependency management system to maximize user productivity; how much productivity will be lost in the event where CoqPIE misses a dependency? First, Coq will not highlight declarations that need to be updated. However, they will still generate errors if the Coqtop process is moved forward over them. An error in a proof script will be missed when moving the Coqtop process as proof scripts are replaced with `admit` to improve efficiency. If the user goes into that script, then the error will be found.

In all cases, these errors can be found by rebuilding the project outside of CoqPIE. We anticipate that these missed dependencies will not be a major impact on productivity as the best mode of operation for proof development is to work top down–that is, to first create the proof statements and then do the proofs. Dependencies that are missed by CoqPIE will most likely mean additional work on the lower levels of a derivation. This work would most likely be done later on even if the errors are known.

Recent work on jedit/PIDE [23] demonstrates a concurrent algorithm to automatically recompile Coq code in the background. This makes detection of errors a little faster but not instantaneous as even concurrent recompilation takes time. Also, the only concurrency that is introduced is that of running proof scripts. The high level compilation of the rest of the source still needs to be done sequentially. However jedit/PIDE will fire off this recompilation automatically where as with CoqPIE it is currently the responsibility of the user to fire off rebuilds regularly.

3.3 High level operations

Certain proof refactoring tasks are common in Coq workflows, and we aim to add explicit tool support for these tasks. Two tasks we help automate include automatically breaking up large theorems into several smaller theorems, and assisting with the modification of proof scripts when replaying. These operations fit well into our tool design because we already have both the ASTs of the sources as well as the parsed Coq output.

3.3.1 Lemma extraction

It is often useful to extract one of the goals of a theorem as a lemma in order to break a large proof into more manageable pieces, and CoqPIE provides a command that automates this extraction. Figure 3.2 shows a sample theorem we want to

attempt an extraction on; CoqPIE can extract a lemma based on the goal state after the selected tactic, which for the example is shown in Figure 3.3. The extraction is done in the following steps:

1. The statement of the new theorem is constructed by taking the goal as the consequent. Each hypothesis becomes an antecedent. If the hypothesis appears to be a variable, then it is encoded as part of a `forall` construct. Otherwise it is encoded as an antecedent of the form *hyp* `->`.

2. The steps used to prove the goal are extracted and become the script for the theorem. One can find the end of the sequence of steps used to prove the current goal at the goal state of each subsequent step. The first step after the current step for which the number of goals is one less than that of the current goal is the last step that needs to be extracted with the theorem.

3. In front of the script from the previous step an `intros` statement is added to introduce all of the generated antecedents.

4. The steps to prove the goal are commented out in the main theorem.

5. An `apply` of the newly generated theorem plus an `apply` for each hypothesis is generated in place of those steps that have been commented out.

6. Finally, note that there is an existential variable in the goal (`?508` in figure 3.3). The lemma extraction tactic tries to figure out how to fill in this variable. The

Figure 3.2: Theorem from which we want to extract a lemma

trick here is to realize that this variable is likely filled in by the steps that prove this goal in the parent theorem. The heuristic is to compare the subgoals after these steps have executed in the main goal to the corresponding subgoals from before they were executed. In this case the `?508` is filled in with `valid (And b (Atom (Const 1))) l = true`.

The result can be seen in Figure 3.4.

This tactic is only a heuristic, and there are several cases in which it will fail. For example, a `Focus` in the middle will break the algorithm for finding the end of the steps for the lemma.

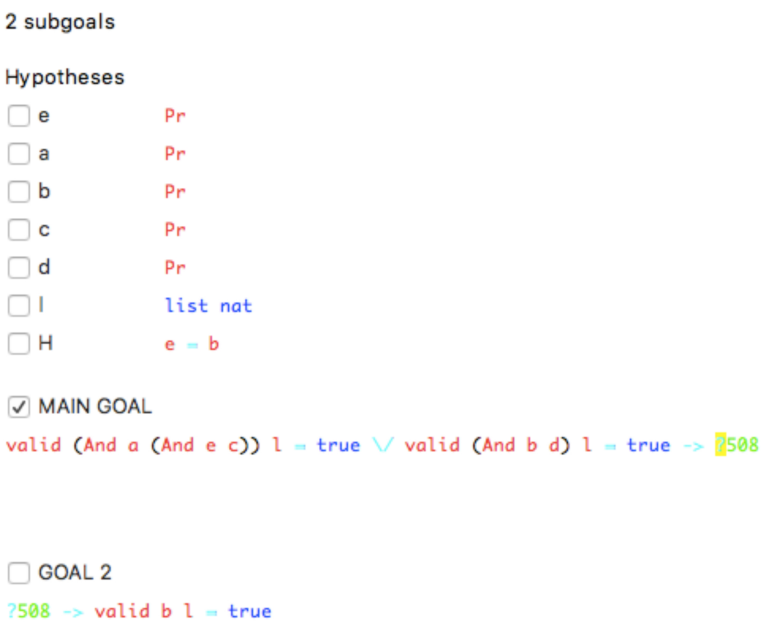


Figure 3.3: Goal state right after executing the selected statement in Figure 3.2.

3.3.2 Replay assist

When the statement of a theorem changes, most of the old proof script may still be correct, but at each step minor changes may need to be made. One common example is that hypothesis names may have changed. For example, `apply H3` may need to become `apply H5`. To improve the workflow we have implemented a replay assistant which automatically will replay proof and apply heuristics to patch the proof back together. Replay assist saves both the coqtop output from before the theorem changed and the output of the new theorem up to the point where a patch may need to be made. One can then compare the two texts and see that `H3` has been renamed `H5`, and patch the proof script accordingly.

To illustrate the replay assistant in action, consider the proof of `popEquiv` in small

```
Theorem newThm : forall e a b c d l, e = b -> valid (And a (And e c)) l = true \/
valid (And b d) l = true ->
valid (And b (Atom (Const 1))) l = true
.
Proof.
    intros e a b c d l H.

    eapply mergeStep.
        eapply Right. eapply Left. eapply Pick. reflexivity.
        eapply Left. eapply Pick. reflexivity. apply H.
    intros. eapply mergeFinish.
Qed.

Theorem mergePredicates: forall e a b c d l,
    e = b ->
    valid (And a (And e c)) l=true \/
    valid (And b d) l=true ->
    valid b l=true.
Proof.
    intros e a b c d l H.
    eapply validTransitive.
apply newThm. apply H.


(*
    eapply mergeStep.
        eapply Right. eapply Left. eapply Pick. reflexivity.
        eapply Left. eapply Pick. reflexivity. apply H.
    intros. eapply mergeFinish.
*)


    intros. eapply and1imply. apply H0.
Qed.
```

Figure 3.4: Theorem and newly generated lemma after extraction

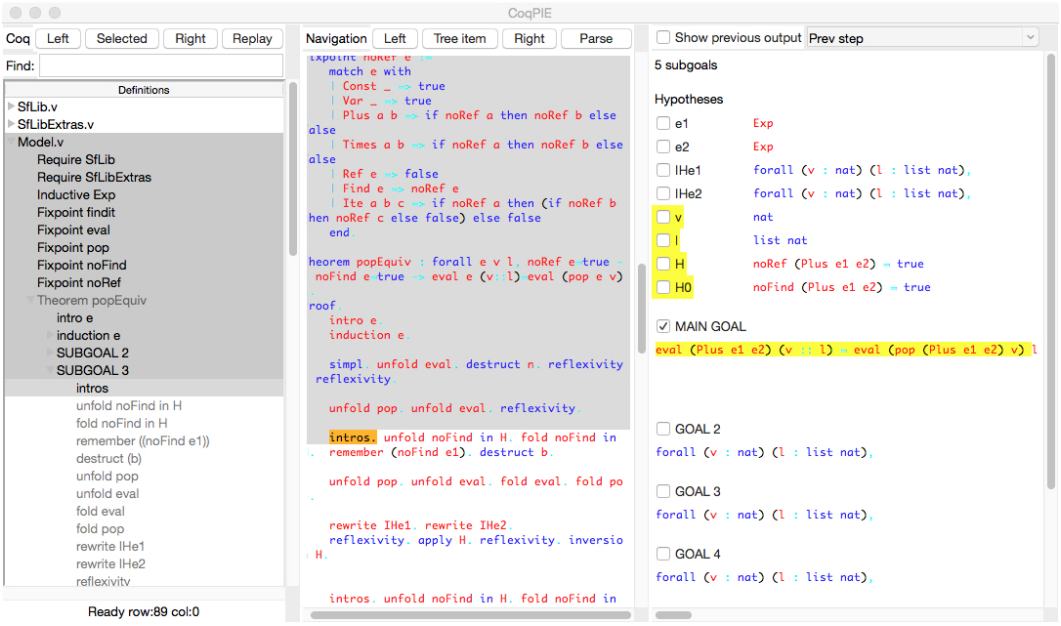


Figure 3.5: The main CoqPIE window after editing definitions and the proof statement but before editing the script.

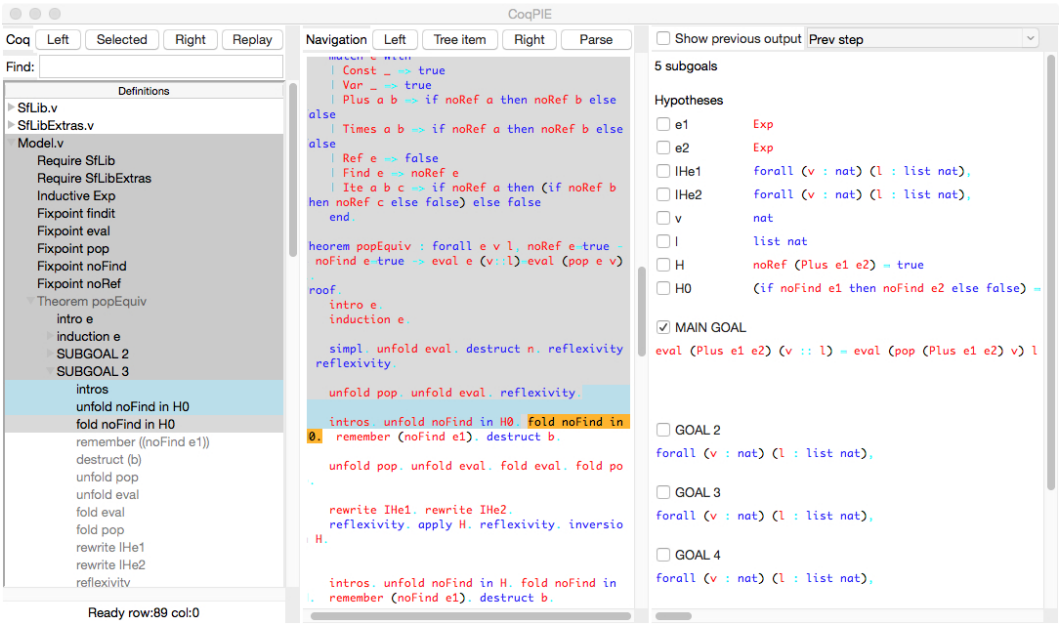


Figure 3.6: The main CoqPIE window after pressing the "Replay" button twice from Figure 3.7.

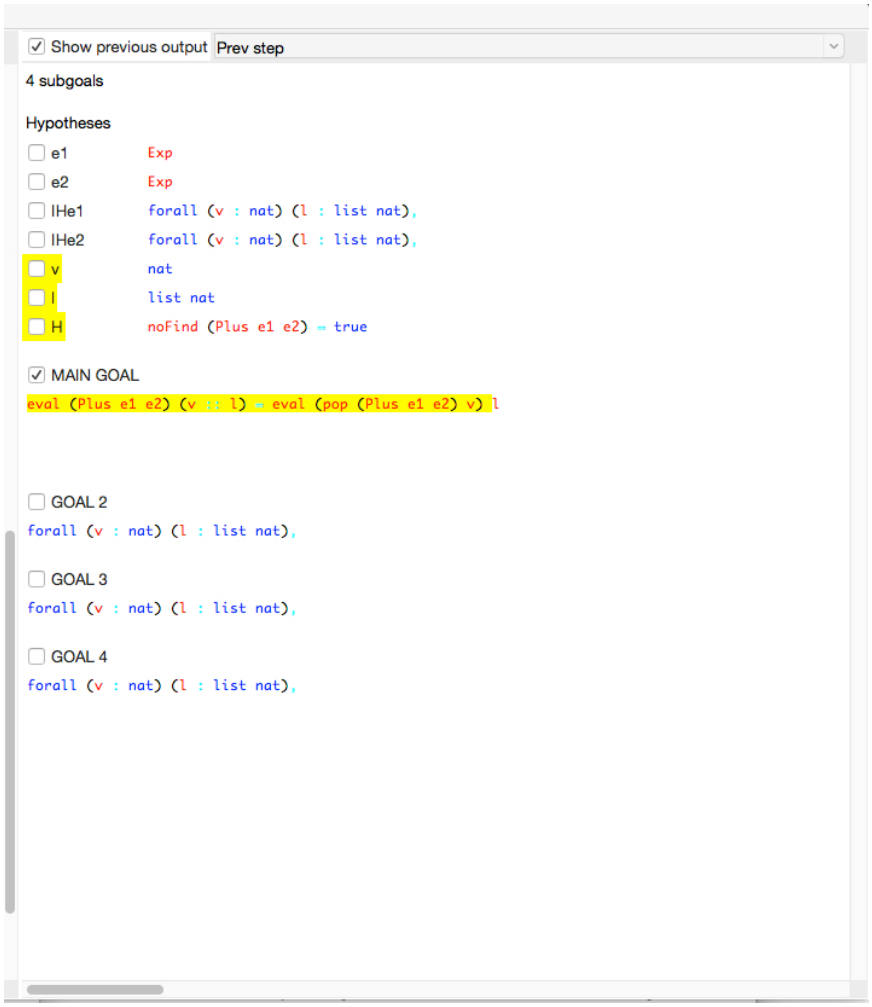


Figure 3.7: The main CoqPIE window when "Show previous output" option is selected.

derivation in `Small/Model.v` (Appendix C) from the repository. There is an extension to an `Exp` data structure is commented out. Consider what happens when the additional `Ref` case is uncommented along with related additions to other definitions. The condition `noRef e==true` also needs to be added to the proof as well; the challenge is to be able to re-use the original proof in the face of these changes. To patch the proof, hypothesis references need to be changed and various sections need to be added to the proof.

The replay assistant provides a semi-automated assistant to help with the task of proof patching. Figure 3.7 shows the state of CoqPIE after all the additions for the derivation in `Small/Model.v` are uncommented as well as the needed change to the proof but before the script is updated. We have rerun the first portion of the script which needs no changes. At this point, the `unfold noFind in H` needs to be changed to `unfold noFind in H0`. We can more clearly see this by clicking the "Show previous output" button to show the old output – see Figure 3.7. Clicking "Replay" twice brings up the state shown in Figure 3.6: the hypothesis reference was automatically patched to the new name by the tool.

The current replay algorithm only makes updates to hypothesis labels. It finds the correct label by looking at both the old and new result from the previous step and choosing the hypothesis from the new state that is the closest match to the one from the old state. Matches are scored by doing a top down comparison of the two AST trees and counting the number of nodes that match.

Coq users will often explicitly name hypotheses that keep changing position during proof development in order to make direct replay more reliable; while this approach improves the odds of a successful replay, the CoqPIE replay tool allows users to skip this step. In addition, we aim to extend CoqPIE replay to support other changes including detecting when a new subgoal has been added, commenting out a subgoal that has been removed, and reordering proof steps. Admittedly it will never be possible to patch back every single proof, but it should be possible to eliminate many of the tedious steps users must take when patching a proof.

3.4 Experience with implementation

The implementation has all of the functionality described in this chapter and is mostly debugged. The author has been using the tool exclusively for proof editing in the DPLL verification described in Chapter 5 consisting of over 10000 lines of Coq code. The tool has also been used to read in a couple of other large derivations including a microprocessor verification example [81][1]and the first few chapters of Software Foundations [65]. We needed to make some minor edits to get Software Foundations to compile. Notably, doubly nested comments cannot be handled currently.

There is an up-front cost of using CoqPIE, the full project needs to be run and intermediate goals parsed and cached. The table in figure 3.8[2] shows times for pro-

[1]A couple of type checking errors showed up in CoqPIE but not when compiling outside of CoqPIE. We are still working to find the source of these errors.

[2]The measurements were taken with versions of the projects from around March 2016

Project	Compile time	CoqPIE initialization time	Memory usage (Python process+ largest Coq process)
Model.v	0:03	0:46	35M+163M
DPLL	1:36	9:08	94M+581M
Microprocessor	3:14	4:19:29	12M+825M
Software Foundations	0:06	4:01	47M+187M

Figure 3.8: Times and memory usage of CoqPIE on different test cases.

cessing some projects from scratch. The times are taken from runs of Coq 8.5pl3 on a 2011 MacBook Pro with a 2.7 Ghz Intel i5 core and 8G of memory. Since this only needs to re-run if the state of the tool becomes inconsistent, it should be an infrequent event; in normal operating mode only the changed information needs to be updated in the cache.

Initialization times for CoqPIE are a few times slower than what is needed to compile the project. This is due to the fact that each file is compiled twice, once to generate a `.vo` file for the project and a second time to capture outputs. Additionally, it takes time to parse the outputs from Coq. While the Python-based editor process uses a nontrivial amount of memory, it is several times smaller than that used by the largest coqtop process and hence the overhead is manageable. While for our current projects the initialization time is tolerable, as shown in the table, for larger projects it will be problematic and we will need to do background updating as is done in PIDE.

3.5 Future work

There are a few critical changes we need in CoqPIE to improve productivity with our DPLL proof.

- Sometimes intermediate goal information gets corrupted and CoqPIE needs to rerun its preprocessing. We plan to allow a variant where the preprocessing can be rerun on a single file rather than the entire project.
- For replay, information on old goals needs to be made persistent across preprocess reruns. Often we replay a proof after rerunning the CoqPIE preproessing step.
- We plan to introduce mechanisms to perform replays across multiple proofs. When the statement in one proof changes, often many lemma proof statements need to change. We plan to develop a macro that can automate all the changes.

There are also a number of areas where improvement we anticipate will be needed for widespread adoption.

- The current implementation requires a recompilation step before the tool can be used. As was described in the previous section, we plan to make recompilation an incremental background process with help of the new PIDEtop interface [13].
- Currently we are using our own parser for the Coq language. We plan to work with the Coq development team to create an API that will allow CoqPIE to

use the built-in parser.

- Company Coq and CtCoq support tools to look up available theorems. Company Coq also has links to Coq's documentation for looking up specific functions and tactics. We plan to integrate these features into CoqPIE.

- We anticipate adding many new rules into the "replay" button to transform tactics for more intelligent replay.

- We are investigating the creation of Python libraries that will allow many of the key features of CoqPIE to be integrated into other IDE tools. People are used to other tools and by making the key features available as extensions to existing tools, we hope to speed up acceptance.

- We plan to add more refactor operations in addition to the lemma extraction described here.

3.6 Related work

In addition to CoqIDE and Proof General, there are several other Coq IDE development efforts. PIDE/jedit [82, 83, 23] introduces asynchronous communication between the IDE and the theorem prover to improve the user experience. The idea is that as text is being edited in a proof script, the theorem prover is continuously running in the background verifying the new text and all dependencies. Concurrency is

used to speed up theorem proving tasks. The tool saves all output and adds markups to the text in appropriate places. Our system does not run the prover as a background task or do automatic updating. However, since PIDE is now integrated into Coq we may add support for concurrency in the future.

CoqPIE provides a goal state window that highlights differences and allows the showing/hiding of individual hypotheses, whereas PIDE/jedit simply stores the text of the theorem prover's output. We do parsing of the output both for the above functionality and replay. CoqPIE does replace proof scripts with `admit` for proofs on which the user is not working. This gains much of the same performance advantage as concurrency. This is because the concurrency in PIDE [23] is based on breaking out scripts within proofs to run on separate processors.

The IDE supplied with the 8.5 version of Coq introduces concurrency and dependency analysis to speed up processing of files. However, these features are not used to create higher level editing commands. While coqtop is processing one can edit portions of the source files not submitted to Coq. There is no project management system, and manual recompilation is needed in order to propagate changes to other files. We currently do not have concurrency in CoqPIE but will likely add it in the future.

Coqoon [2] is an effort to integrate Coq into Eclipse. It provides a tree view to show all files and declarations in the Coq input, similar to our tree view. Parsing is less developed than what exists in CoqPIE. Cocoon provides a simple lexer for tokens

and determines the dividing point between definitions by finding periods. CoqPIE on the other hand provides full AST generation along with links between the nodes and positions in the text. Additionally, we parse goal output along with the source code. There is no concept of storing both the old an new versions of goals in Coqoon and hence no framework for the style of replay assist provided by CoqPIE. Since there are no ASTs, refactoring operations such as lemma extraction are not done. Finally, there is no difference highlighting since it is also dependent on having a full AST. Coqoon is built on top of PIDE and so it does allow for asynchronous recompilation of proofs. The PIDE protocol also allows Coqoon to have cached output at each step. The initialization process to pre-populate annotations is not needed; instead, theorem proving is a background task and annotations are collected as they become available.

There also are efforts to build Coq IDEs at MIT and UCSD [4, 3]. Both are web-based. However, these tools are primarily intended for teaching.

Proviola [80] has developed a tool that compiles Coq source code and captures the output at each step. The tool then generates a Javascript-based web page that can display the outputs as the user hovers over each tactic in a proof. Our tool in addition to caching output also parses the output so it can be used by editing macros. CoqPIE also provides algorithms for updating the cache when the source code is edited and the Coq process is rerun.

Pcoq [17] is an earlier UI for Coq. It features a window showing the proof script, another window showing the Coq output and a third window showing a list of po-

tential theorems that can be applied at the current step. The first two windows are similar to what exists in Proof General and Coq IDE. The third window is unique to Pcoq and would be a useful feature to add to CoqPIE. CtCoq [84, 46] builds on Pcoq. It provides the same basic windows as Pcoq, and also parses Coq syntax. It is integrated directly with the Coq AST data structure. Unlike CoqPIE, this AST parsing is used to create a tree-oriented editing paradigm. UI-based point/click/drag and drop commands are used for constructing proofs in place of entering commands. In comparison, our system uses the ASTs to implement complex operations such as replay assist and lemma extraction.

Company Coq [67] is an extension to Proof General that adds many useful features, including shortcut text entry, completion, and reference to Coq documentation. These features would also be useful to add to CoqPIE but they are not part of our primary focus. Company Coq also includes a lemma extraction feature. However, its implementation is not based on using ASTs for the proof script or goal outputs; simpler heuristics are used to extract the text needed for the lemma. For example, a command is sent to Coq to print out hypotheses which Company Coq then captures. The CoqPIE lemma extraction, hence, is more accurate but not perfect.

Proof script transformations have been discussed in [45]. Their method involves creating a few correctness preserving transformations. Since the transformations must be formally verified it limits the scope of what tasks can be performed. The refactoring operations in CoqPIE are heuristic in nature so correctness all falls back on Coq, and

the user may need to edit inaccurate tool results by hand. Hand-editing to patch the result should be less work than doing the complete transformation manually.

3.7 Conclusion

We have presented CoqPIE, a novel Coq editing framework. The key feature of CoqPIE is use of an integrated parser that links AST nodes to source text. This allows us to create several different forms of intelligent editing functionality, including proof refactoring, showing differences between terms to help guide proof development, and maintaining dependencies so that out-of-date information is clearly highlighted.

Most of the features have been useful in our development of DPLL. The one exception is replay. The problem here is persistence. Most often replays are done after a rerun of the CoqPIE preprocessing and the old state information is lost. We anticipate it will be useful when this issue is resolved.

The current implementation develops a few refactoring tools, but we have only scratched the surface of what refactoring tools can be built over the CoqPIE foundation. To facilitate acceptance of these features, in addition to the stand alone editor, we plan to develop libraries that can be integrated into other popular IDEs.

Chapter 4

The Advanced Rewriting Plugin

Our PEDANTIC framework (Chapter 2) integrates a customized simplification algorithm for separation logic assertions. While quite useful, the code is quite slow. In this chapter, we describe an ML-plugin which is much faster and provides a more general framework for simplification. ML plugins are libraries that link directly into the Coq theorem prover and can work directly on the Coq data structures.

Simplification is one of the most important pieces of any interactive theorem prover. The stronger the mechanism, the more usable the tool will be. The speed of a simplification algorithm also plays an important role in the usability of a theorem prover. A user expects simplification to consistently finish within a few seconds. Coq's [58] simplification mechanism is limited. The `simpl` tactic, for example, does little more than symbolic computation and becomes quite slow on large expressions. Isabelle [62] provides a more powerful simplification mechanism. Simplification is

considered to be one of the main advantages of Isabelle over Coq. It is also often observed that many intuitively obvious facts are difficult to prove in a theorem prover such as Coq using the existing tactics.

The good news is that Coq provides the ability to add new tactics through an ml-plugin mechanism. Plugins have already been introduced for AC (associative/commutative) rewriting [22] and SAT/SMT solving [11, 41].

In this chapter, we introduce the advanced rewriting plugin, a plugin for Coq that gives the theorem prover a simplification tactic which is much more powerful than what is available with `simpl` and in many cases what is available in Isabelle. In particular, our system provides better AC matching algorithms and enforcement of recursive path orderings [32, 36, 47] to ensure termination. The tool has many builtin simplifications for common operators such as arithmetic and boolean operators and there also are rules for simplifying quantifier formulae. In addition, the user can add conditional rewrite rules for other operators. The rewriting package described in this chapter also automatically verifies intuitively obvious goals. This is discussed more in Section 4.3.

While the basics of simplification and term rewriting is a mature topic, this system introduces several important innovations:

- Recursive path orderings are used to provide a unifying framework to integrate many different simplification algorithms and conditional rewriting. The general idea is that if an algorithm produces a simplified version of an expression, then

the rewrite is accepted, if not, then the algorithm fails. Rewriting terminates when no algorithms or rewrite rules can be applied.

- AC matching and rewriting is deeply embedded into the matching algorithms. Coq expressions are converted to a custom data structure used by the advanced rewriting library. When simplification is finished, the expressions are converted back to Coq's representation. In the advanced rewriting library data structure, AC operators are flattened: $(a + (b + c)$ becomes $+(a, b, c))$. The pattern matching algorithms then automatically switch the ordering of parameters where necessary.

- Transitive closure computations can be used to simplify expressions with inequalities. For example, our system will simplify $x < y \wedge y < z \wedge z < x$ to `False`.

- Every subexpression seen by the system is integrated into a single giant DAG. When a new expression is parsed or generated from rewriting, the system adds it to the DAG working up from the leaves. Each subexpression gets a unique *intern number* which is used to cache useful information about the subexpression.

As a final contribution of the chapter, when comparing the data structures used to represent terms in our system to those in Coq, we found three optimizations that can benefit Coq. 1) All subterms should be interned into a single DAG as described above and caching should be implemented. 2) Symbols should be interned (thus replacing

string comparisons with integer comparisons). 3) Coq internally represents naturals using a unary representation. This should be replaced with a bignum representation.[1].

4.1 An overview of rewriting capabilities

In this section, we give a description of each of the rewriting algorithms in our library. Our rewriting system is composed of many smaller algorithms, and each is required to produce a result that is smaller than its input (or, to produce no result). We use recursive path orderings [32, 36, 47] to form a partial ordering among expressions. This provides the basis for integrating rewriting algorithms.

The next several sections detail the various algorithms used in our tool.

4.1.1 Inner rewriting

All of the rules and algorithms are applied in an innermost fashion. For a term $f(t_1, \ldots, t_n)$, first each of the subterms are simplified yielding $f(t'_1, \ldots, t'_n)$. Then the simplifications are applied to the term as a whole. If the top level term is simplified, then the whole process is repeated. Recursive path orderings ensure that the process terminates.

[1]This third item is actually being done in Coq 8.8

4.1.2 Rewrite rules

Rewriting rules/conditional rewriting rules form the basis of our system. Rewrite rules replace one subterm with a simpler one. For example, the rule

$$x + 0 \to x$$

is built into the system. We can also have conditional rewrite rules which are of the form $l\{c\} \to r$. If the term c rewrites to `True`, then the rule can be applied. For example, one might introduce the following conditional rewrite rule

$$a * x > 0\{x > 0\} \to a > 0$$

4.1.2.1 Multiple matches

Sometimes multiple rules will match the term being simplified. When this is the case, one rule is chosen arbitrarily. The assumption is that all paths will lead to a single simplest state (though the system does not guarantee this). Builtin rules for simplification algorithms will generally be executed first before user algorithms.

4.1.3 Termination

Recursive path orderings [32, 36, 47] are used to ensure termination. The basic idea is that we first assign a partial ordering among function symbols, a precedence

ordering. Usually more expensive functions have higher precedence. Also, if a function f uses g in its definition, then f has higher precedence than g. In general, a term $t = f(t_1, \ldots, t_m)$ is bigger than a term $u = g(u_1, \ldots, u_n)$ if

- u is a subterm of t,
- if $f > g$ and t is greater than all $u_i \in \{u_1, \ldots, u_n\}$,
- if $f = g$, f is not AC, $m = n$ and each t_i is bigger than the corresponding u_i or
- $f = g$, f is AC, and for each u_j, there is a t_i which is bigger.

One of the interesting properties of the ordering is that it is closed under substitution, if $t > u$, then for any substitution θ, $t\theta > u\theta$. The implication of this is that for any rewrite rule $l\{c\} \to r$, if $l > r$, then any term x rewritten by the rule to x' will be smaller (ie $x > x'$). We also require $l > c$. We test the condition of the rewrite rule by doing a substitution and then recursively calling our rewrite system to simplify.

Many extensions to the basic concept such as lexicographic orderings [47] have been developed.

4.1.3.1 Exponential explosions

In addition to termination, it is also necessary to prevent other exponential explosions when rewriting. There is no strictly enforced rule. However, the general rule of thumb is that for any rule $l\{c\} \to r$, any variable should only appear once in either c or once in r. The reason for this is that a single variable may be replaced with fairly

complex expression when applying the rule. Duplicating that expression may cause an exponential explosion in the work that needs to be done. Consider what would happen with the following rules:

```
lsub(Nil,b) -> b
lsub(a,Nil) -> a
lsub(Cons(a,b),Cons(c,d))
    { length(lsub(b,Cons(c,d))) < length(lsub(Cons(a,b),d)) }
    -> lsub(b,Cons(c,d))
lsub(Cons(a,b),Cons(c,d))
    { not(length(lsub(b,Cons(c,d))) < length(lsub(Cons(a,b),d))) } ->
    lsub(Cons(a,b),d)
```

If you call `lsub` with a couple of long lists, you will have a fairly expensive computation– even when all intermediate results are cached.

4.1.4 Builtins for common operators

Many builtin simplifications are just rewrite rules stored in a library such as $x + 0 \rightarrow x$. However, there are also algorithms that are slightly more complex. For example, there are algorithms that can simplify linear equations. For example, $3 * x = 9$ can be simplified to $x = 3$. However, the algorithm needs to check that the 3 and 9 are indeed constants and that 3 divides 9 evenly. This cannot be done with a rewrite rule. One would have to write something like:

```
c1 * x = c2 { cond } -> x = c1 / c2
```

The problem is that there is no way of writing a `cond` that can check that `c1` and `c2` are constants rather than other expressions and that `c2` divides `c1` evenly.

4.1.5 Contextual rewriting

Often when simplifying a subterm in an expression, one can gain useful information from another subterm in that same expression. Consider simplifying the expression $x = 0 \wedge x + y = 3$. This simplifies to $x = 0 \wedge y = 3$. In order to do this, we need to assume that $x = 0$ when simplifying $x + y = 3$. The idea is to add rewrite rules when simplifying $x + y = 3$. To do this, we need to introduce a system of marked variables (denoted with x'). These marked variables in a rewrite rule match a variable directly. For example, x' in l of a rule $l\{c\} \rightarrow r$ matches the variable x and not the variable y or any other variable. Our system will introduce the rule $x' \rightarrow 0$.

Contextual rewriting is implemented within the top level inner rewrite loop. Figure 4.1 outlines the top level algorithm. The function recursively traverses terms rewriting expressions. If the function is a conjunction, disjunction, if-then-else or other construct that introduces context, then the function `add_rule` is used to add appropriate rules to the context before simplifying the subterms. Derived rules (section 4.1.5.1) are used to convert these rules to more usable forms. The function `rewrite_top` is used to rewrite the term at the top level. First, algorithms for algebraic simplification, quantifiers, lambdas, transitive rewrites (as described in later sections) are applied. If no simplification algorithm is applicable, then `rewrite_top` searches for rules that match the term. The first one found is applied.

```
rewrite_term(env,f(t_1,...,t_n)) :=
 if f =' &' then /* AC flattened conjunction t_1 & ... & t_n */
      /* rewrite each t_i in a context where each t_j is
       asserted if i ≠ j */
      loop
       orig = '&'(t_1,...,t_n)
       for i=1,...,n
        env' = env
        for j=1,...,n do
         if i ≠ j then
          env' = add_rule(env',t_i → true)
        t'_i = rewrite_term(env',t_i)
       res = rewrite_top(env,'&'(t'_1,...,t'_n))
      until res=orig
      return res
 else if f =' ||' then /* AC flattened disjunction t_1||...||t_n */
      /* rewrite each t_i in a context where each ¬(t_j) is
       asserted if i ≠ j */
      loop
       orig = '||'(t_1,...,t_n)
       for i=1,...,n
        env' = env
        for j=1,...,n do
         if i ≠ j then
          env' = add_rule(env',t_i → false)
        t'_i = rewrite_term(env',t_i)
       res = rewrite_top(env,'||'(t'_1,...,t'_n))
      until res=orig
      return res
 else if f ='if' then /* if t_1 then t_2 else t_3 */
      t'_1 = rewrite_term(env,t_1)
      env' = add_rule(env,t'_1 → true)
      t'_2 = rewrite_term(env',t_2)
      env'' = add_rule(env,t'_1 → false)
      t'_3 = rewrite_term(env'',t_3)
      return 'if t1 then t2 else t3'
 ...
 else // Default case
      loop
       orig = f(t_1,...,t_n)
       for i=1,...,n
        t'_i = rewrite_term(env,t_i)
       res = rewrite_top(env,f(t'_1,...,t'_n))
      until res=orig
      return res
```

Figure 4.1: Pseudo-code for contextual rewriting algorithm

4.1.5.1 Derived rules

The next question is how the system decides on the rule $x' \to 0$ when simplifying the right conjunct of $x = 0 \wedge x + y = 3$. This is done in two steps. First, the rule $x' = 0 \to \texttt{true}$ is generated. This is done by simply marking the variables in the term. The second step is done through the use of generating a derived rule. The following transformation meta-rule is used, $x' = t \to \texttt{true} \Rightarrow x' \to t$ where t is a constant term such as a number, `true`, `false` or a constructor all of whose parameters are constant terms such as $\texttt{Cons}(1, \texttt{Nil}())$. Our system uses recursive path orderings to orient rules that are generated from equalities. In the case above, we have made a special augmentation to the recursive path orderings to recognize constants as being smaller than variables. As another example, in our system, the expression $f(x) = x \wedge f(f(x)) = x$ will simplify to $f(x) = x$. When the system simplifies $f(f(x)) = x$, it will do it in the context of $f(x) = x$ because x is smaller than $f(x)$ with respect to the recursive path ordering, the system will introduce the rewrite rule $f(x) \to x$. Applying this rule twice to the expression $f(f(x)) = x$ will yield $x = x$. This simplifies to `True` and eliminates the term.

4.1.6 Quantifiers and their rules

Our system contains many specialized rules for simplifying expressions that involve quantifiers. As an example there is a rule to propagate negation inside a universal

or existential quantifier, $\texttt{not}(\forall x.t) \rightarrow \exists x.\texttt{not}(t)$. There are also some more complex rules. For example, the expression $\exists x.x = t \wedge e$ can be simplified to $e[t/x]$ when t does not contain the bound variable x.

4.1.7 Lambda and higher order functions

Our rewrite system supports higher order reasoning by introducing an `apply` operator, a lambda quantifier and the rewrite rule $\texttt{apply}((\lambda x.t), e) \rightarrow t[e/x]$.

4.1.8 AC functions and pattern matching algorithms

Our rewriting framework provides extensive support for functions with associative and/or commutative properties. First, any expression involving an AC operator is flattened. For example, $x + (y + z)$ is converted to $'+'(x, y, z)$.

Our rule matching algorithm has special match handling for AC properties. If one tries to apply the rule $x + f(y) \rightarrow g(x, y)$ to the term $f(q) + h(r)$, the result will be $g(h(r), q)$. The match algorithm automatically switches the order of the parameters. Now consider what happens if we match the above rule to the expression $f(x)+h(y)+i(r)$. The application can produce three possible results, $g(h(y)+i(r), x)$, $g(h(y), x) + i(r)$ or $g(i(r), x) + h(y)$. The basic rule here is that the variable x in the rule can be matched to one or more terms in the expression. If it is matched to more than one term, then the terms are combined with the AC operator (in this case '+').

The result of our special handling for AC operators is that not only can multiple rules fire on a single expression, but a single rule on its own may produce multiple results. As a simplification to limit the number of matches, if a rule has more than one variable in an AC operator, only one can be matched to multiple terms. For example, when using using the rule $x + y + q(z) \rightarrow qq(x, y, z)$, either the variable x or y can match multiple terms but not both. This means for example, that when simplifying the term $a + b + c + d + q(e)$ with the above rule, we will not generate the result $qq(a + b, c + d, e)$.

One limitation of our implementation is that it does not support neutral elements (or identity elements). For addition, this would be the neutral element is the term 0. For example, if we have a rule $f(x) * 2 + y \rightarrow g(x) + y$, our current implementation cannot simplify the expression $f(x) * 2$. To do this, y would need to be matched to a neutral element 0. This is an extension we may make in the future. However, most useful cases are actually covered by having a rule of the form $x + 0 \rightarrow x$ in the system.

4.1.9 Equalities, total orders and partial orders

Our system provides mechanisms to compute transitive closures in simplifying expressions. For example, the expression $a < b \wedge b < c \wedge c < a$ will be simplified to `false`. The transitive closure reasoning is embedded within contextual rewriting. The idea here is that when simplifying $a < b$, there will be two contextual rules $b' < c' \rightarrow$ `true` and $c' < a' \rightarrow$ `true`. From these two rules, a third rule is produced

$b' < a' \rightarrow$ `true`. The transitive closure reasoning then recognizes that having $b' < a' \rightarrow$ `true` means we can replace $a < b$ with `false`. This gives us `false`$\wedge b < c \wedge c < a$. By applying other rules, this is simplified to `false`.

When computing transitive closures our system will insert inequalities for constants. For example, if both the constants 3 and 5 are detected within the expression, then the inequality $3 < 5$ will be generated and inserted before computing the transitive closure. This allows us to reduce an expression such as $5 < x \wedge x < y \wedge y < 3$ to `false`.

4.2 Implementation

In this section we discuss a few of the details of our implementation.

4.2.1 Expression representation and Interning

Figure 4.2 shows the data type for expressions. One should note that many of the places in which a function or variable name is needed, the type is `int`. This is because all symbols are stored in an intern table. This makes comparisons when matching more efficient.

The `VAR` case represents variables (free or bound). The `MARKED_VAR` is a variable that is primed in a contextual rewriting rule. For example in the rewrite rule $x' \rightarrow 0$ from Section 4.1.5, the x' is a marked variable. The `QUANT` case represents a quantifier.

The first parameter is the type of quantifier `ALL`, `EXISTS` or `LAMBDA` (represented by an interned string). The second parameter is a list of the bound variables (name and type). The third parameter is the expression inside the quantifier. The fourth parameter is only used for `ALL` and is a further guard on bound variables. For example, we can have an expression `ALL`$(x : x < 3)x + y > 5$. The $x < 3$ is a guard. The `LET` case is a let expression. The first `exp` can be either a variable or constructor pattern. Then we have the type. The next `exp` is the value assigned to the variable (or matched to the constructor pattern). The final `exp` is the expression to evaluate with the newly bound variable (or variables). `CASE` is similar to `LET`. The first `exp` is the expression we are matching. The second parameter is its type. Third we have a list of expression pairs. The first expression in the pair is the pattern, the second is what to evaluate if the pattern matches. We then have cases for constants. `NUM` is an integer constant. `RATIONAL` is a rational number. `STRING` is a string. `CHAR` is a char constant. `NOEXP` is no expression.

4.2.1.1 Interning of subexpressions

As an optimization, all subexpressions are interned. All expressions are organized into a giant DAG. Each subexpression is a single node and appears only once in the DAG. Each subexpression is assigned a unique integer identifier. The `REF of int` case of the type in Figure 4.2 represents the interned version of an expression. As an example, a sub-expression such as `x+y` is assigned a unique intern value (such as 25).

Whenever this expression is encountered in computation, the same intern value will be returned.

4.2.1.2 Efficiently looking up intern values for subexpressions

In order to make interning of subexpressions practical, one needs an efficient algorithm to intern an expression. The good news is that with the exception of when an expression is parsed, most of the time it will only be necessary to intern a top level expression given the interned form of all its subterms. The key to doing this efficiently is that a reverse index needs to be stored. Given a set of subterms (and a functor), the index needs to return the intern number for the parent expression if it exists. In our system, we simplified this index a bit. Instead of indexing all subterms, we have an index from any subterm to all parents and then we search for the appropriate parent.

4.2.1.3 Interning of AC subexpressions

Subterms produced with AC functors (such as `+(x,y,z)`) are interned by first sorting the arguments to the AC operator. This causes the expressions `+(x,y,z)` and `+(z,y,x)` to have the same intern value. It also causes the side effect that if the user enters the expression `z+y+x`, it may print out as `x+y+z` as information on the order of the parameters is lost.

4.2.1.4 Equality test

One interesting result of the interning of subexpressions is that test for equality between subexpressions reduces to testing the equality between two intern numbers.

4.2.2 Caching of rewrite results

When computing the results of rewriting, its simplified form is cached. Hence, rewriting only needs to be computed once. This means that the Fibonacci rules shown in figure 4.3 will exhibit linear complexity despite not being written with memoization. The key to implementing the cache are the intern numbers. The cache is represented as a mapping from integers to integers (ie from the intern number of the expression to the intern number of the result after rewriting).

4.2.2.1 Contextual rewrite cache

Rewriting may produce different results when done within the context of another expression. For example, simplifying the subterm $x + y$ in the expression $x + y = z \wedge x = 0$ will yield $y = z$ whereas simplifying $x+y = z$ on its own will yield $x+y = z$. When rules are added to the context, to find (or save the cached rewrite), we first generate a term $\texttt{CC}(t, \texttt{rules}(r_1, \ldots, r_n))$ where `rules` is registered as an AC operator and $r_1, \ldots, r_n$ are all of the rules in context sorted by intern number. We then save the result of rewriting under the intern number produced for the above expression.

```
type exp = VAR of int
         | MARKED_VAR of int
         | QUANT of (int * (int * Rtype.etype) list * exp * exp)
         | APPL of (int * exp list)
         | LET of (exp * Rtype.etype * exp *exp)
         | CASE of exp * Rtype.etype * ((exp * exp) list)
         ...
         | NUM of int
         | RATIONAL of int * int
         | STRING of string
         | CHAR of char
         | REF of int
         | NOEXP ;;
```

Figure 4.2: AST for expressions

```
fib(Z)          →  1
fib(S(Z))       →  1
fib S(S(n)))    →  fib(S(n))+fib(n)
```

Figure 4.3: Rules for Fibonacci numbers

For example within the expression above, we generate $\texttt{CC}(x + y, \texttt{rules}(x \to 0))$. The result of the rewriting (y) is saved under the intern number for this expression.

4.2.2.2 Issues of interning and decoding expressions

Throughout the code for the advanced rewriting library any function that takes a parameter of type `exp` needs to deal with the `REF` case from figure 4.2. The infrastructure provides functions for converting these terms back into expressions using the other constructs in figure 4.2. There are three such functions, `decode_exp`, `decode_one_exp` and `decode_two_exp`. The first removes all occurrences of `REF` from the expression. The second only removes the `REF` it is the root of the expression. The third will only remove `REF`s that are either the root or one of the parameters to the

root expression. Since there are many functions in the advanced rewriting library that deal with expressions, there are a large number of calls to the above functions. Unfortunately, it turns out there is no one place to cleanly do the interning and decoding. Also, if a function returns an expression as its result, the result is interned so as to be cached. There is an `intern_exp` function which interns an expression. The result of `intern_exp` will always be of the form `REF` n.

4.3 The interface to Coq

The interface[2] consists a single tactic `arewrite`. This tactic is similar to `simpl`. The current goal is replaced with one that is simplified. There is also a variant `arewrite in H` to simplify a hypothesis `H`.

4.3.1 Soundness

One of the key issues with our system is ensure that rewrites are sound with respect to Coq's logic. After rewriting is performed, the result is integrated into the proof tree using Coq's replace tactic. This generates two subgoals. The first is the rewritten version of the original goal. The second goal is of the form `orig=rew` which is used to justify the rewrite as shown below:

```
============================
1 + 1 = 3
```

[2]Work on the interface was not fully complete at the time the thesis was submitted. However, the author expects this work to be done very shortly after the thesis is submitted.

```
x < arewrite.
2 subgoals

 ============================
 False

subgoal 2 is:
False = (1 + 1 = 3)
```

Currently, the rewriter simply produces the second goal. However, our rewriter will eventually have tools to construct the proof of the second goal using Coq tactics to verify soundness. This might be slow. As such, we may include an option to turn off this proof generation (and simply stick in an `admit`) to improve interactive performance.

4.3.2 Hypotheses

When `arewrite` is used to simplify a goal, all hypotheses are added to the rewriting context. For example if one has the following goal,

```
H: x < 3
----------------------
x > 4
```

`arewrite` will simplify the `x > 4` to `False` using the hypothesis `H`. Because hypotheses are automatically added to the context, `arewrite` could potentially be developed into a more general purpose tactic for proving intuitively obvious goals. Note that when `arewrite in H` is used to simplify a hypothesis, no other hypotheses are added to the environment.

4.3.3 Environment information

As part of the `arewrite` tactic, the coq interface picks up information from the theorem proving environment to assist in rewriting.

4.3.3.1 Extraction of precedence information from Coq

Symbol precedences play a key role in the recursive path partial ordering of terms. Recursive path orderings are used to decide the orientation of rules when a term is introduced for contextual rewriting. For example, if one is simplifying the term $\texttt{f(x)} = \texttt{g(x)} \wedge \texttt{g(3)} = \texttt{g(4)}$, when simplifying `g(3)=g(4)`, if `f` has smaller precedence than `g`, then the rule $\texttt{g(x)} \rightarrow \texttt{f(x)}$ will be introduced. Most precedence information can be extracted automatically. If a function f is used in the definition of a function g, then we add the precedence $f < g$.

4.3.3.2 Extraction of rewrite rules from Coq definitions

Many function definitions can be fairly easily be converted into rewrite rules. The main constraint is that the rule must follow recursive path orderings constraints. Otherwise the rule will not be automatically generated. If the definition uses either a `match` or `if-then-else`, then multiple rules will be generated (one for each branch). Match variables will be appropriately instantiated in the rules and for `if-then-else` conditional rewrite rules will be generated. Here are a couple of examples to illustrate how this will work:

```
fixpoint append(a) =
    match a with
    | nil => b
    | cons f r => cons f (append r b)
    end.
```

will generate:

$$\texttt{append}(\texttt{Nil}, b) \to b$$
$$\texttt{append}(\texttt{Cons}(f, r), b) \to \texttt{Cons}(f, \texttt{append}(r, b))$$

and

```
fixpoint fact(a) =
    if n=0
    then 1
    else fact(n-1)*n.
```

will generate:

$$\texttt{fact}(n)\{n = 0\} \to 1$$

Notice we do not get the second rule

$$\texttt{fact}(n)\{n \neq 0\} \to n * (\texttt{fact}(n-1))$$

This is because the rule does not follow the constraints of recursive path orderings.

4.4 Results

While our integration of the rewriting library into Coq is relatively new, we do have some promising initial results that we now detail.

4.4.1 Comparison of different rewriting librarires

To compare the capabilities of the various prover rewriting systems, we have produced the table in Figure 4.4 of specific examples comparing our Advanced Rewriting Library with Coq's built-in simplifier and Isabelle's simplifier.

The first two examples show how the advanced rewriting library can incorporate one operand of a conjunct into context while simplifying the other. For the first sample, the right operand $f(f(x)) = x$ is simplified twice with $f(x) \to x$ from the left operand to yield $x = x$. This operand is then removed leaving just $x = f(x)$. The second example shows how $f(f(f(x)))$ is simplified with $f(f(x)) \to x$ to yield $f(x) = x$. This is then used to simplify $f(f(x)) = x$ to $x = x$ and that term is eliminated leaving only $f(x) = x$. The third example shows how the right side of in implicant is rewritten using the left hand side. The result here is that $x + 1 = 4$ is rewritten to $3 + 1 = 4$ which simplifies the whole implicant to `True`. The next three are variants. When $x + 1 = 4$ is on the left of the implicant, it is first simplified to $x = 3$ and then used to simplify the right hand side. The seventh expression in the figure shows how an existential can be eliminated if part of its operand assigns a value to the quantified variable. Examples 8 and 9 show how transitive closures can be used to simplify expressions. Example 10 shows how the advanced rewriting library performs algebraic simplifications. The last example shows how AC rewriting is used. $x + y \to 0$ is generated from the left hand side expression. It is used to simplify $y + 3 + x$ to $0 + 3$ or just 3.

	Expression	Coq's `simpl`	Isabelle[3]	Advanced Rewriting Library
1	$x = f(x) \wedge x = f(f(x))$	$x = f(x) \wedge x = f(f(x))$	fails	$x = f(x)$
2	$f(f(x)) = x \wedge f(f(f(x))) = x$	$f(f(x)) = x \wedge$ $f(f(f(x))) = x$	fails fails	$f(x) = x$
3	$x = 3 \rightarrow x + 1 = 4$	$x = 3 \rightarrow x + 1 = 4$	solved	`True`
4	$\forall x, x = 3 \rightarrow x + 1 = 4$	$\forall x, x = 3 \rightarrow x + 1 = 4$	fails	`True`
5	$x + 4 = 1 \rightarrow x = 3$	$x + 1 = 4 \rightarrow x = 3$	fails	`True`
6	$\forall x, x + 1 = 4 \rightarrow x = 3$	$\forall x, x + 1 = 4 \rightarrow x = 3$	fails	`True`
7	$\exists x. f(x) = g(4) \wedge x = 3$	$\exists x. f(x) = g(4) \wedge x = 3$	$f(3) = g(4)$	$f(3) = g(4)$
8	$x < y \wedge y < z \wedge z < x$	$x < y \wedge y < z \wedge z < x$ `omega` fails as well.	fails	`False`
9	$5 < y \wedge y < z \wedge z < 3$	$x < y \wedge y < z \wedge z < x$ `omega` fails as well.	fails	`False`
10	$\forall x, 2 * x + 1 = 7$	$\forall x, x + (x + 0) + 1 = 7$	fails	$\forall x, x = 3$
11	$x + y = 0 \rightarrow$ $y + 3 + x = q$	$x + y = 0 \rightarrow$ $y + 3 + x = q$	fails fails	$0 = y + x$ $\rightarrow 3 = q$

Figure 4.4: Comparison of rewriting systems on different expressions

Based on the experimental results in the table, we conclude that Isabelle has a reasonable system for performing contextual rewriting. It succeeded on the third and seventh test cases. It does not do algebraic simplifications. Test cases 5 and 10 demonstrate this. There is no transitive reasoning in either Coq or Isabelle as demonstrated by cases 8 and 9. Coq's `omega` tactic also failed on these examples. Test case 11 shows that AC matching in Isabelle is limited. We also note that Coq's `simpl` tactic is limited as it failed on all of these test cases. We should note that some algebraic simplification can be done with Coq's `ring_simplify` tactic. However, it works with rationals and not the default natural number type.

4.4.2 Experience with PEDANTIC

A simplified version of this rewriting system is implement using Coq's Gallina as part of our PEDANTIC framework described in Chapter 2. The integration is in its early stages and we do not have extensive experience with the use of the system. The rewriting in PEDANTIC plays a key role in bringing states into a canonical form. However, it does not do all of the work. User intervention is still needed for some of the steps. One of our key motivations in developing this library is to improve the performance of simplification in PEDANTIC. Simplification takes over 80% of PEDANTIC's CPU time.

Let's look at a sample sequence of rewrites to illustrate how contextual rewriting plays a key role in PEDANTIC. When we discussed reasoning about the tree traversal in Chapter 2 we omitted details on how states are simplified. We show how some simplification is done for the State 2.4.8 from Chapter 2. This simplification is done after unfolding but before the magicwand is eliminated.

PEDANTIC's simplification will simplify the ALL construct. First, the $v_0 \in \texttt{TreeRecords}([v_6, v_7, v_8])$ is simplified to $v_0 = v_6 \vee x \in \texttt{TreeRecords}(v_7)$. Further simplification of the ALL splits it into two predicates. We also simplify the $nth(v_6 :: v_7 :: v_8 :: nil, 1)$ and $v_6 + 0$ yielding the separation logic expression in State 4.4.1.

State 4.4.1

$$
\begin{array}{l}
(\texttt{AbsExists}\ v_8\ (\texttt{AbsExists}\ v_7 \\
\quad (\texttt{AbsExists}\ v_6\ (\texttt{AbsExists}\ v_5(\texttt{AbsExists}\ v_4 \\
\quad\quad (\texttt{AbsExists}\ v_3\ (\texttt{AbsExists}\ v_2(\texttt{AbsExists}\ v_1 \\
\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ v_6 \\
\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{T}\ (v_6+1) \\
\quad\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_r}\ (\texttt{T}+1) \\
\quad\quad\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ \texttt{T} \\
\quad\quad\quad\quad\quad\quad\quad (\texttt{P}+1 \mapsto \texttt{T} * \texttt{P} \mapsto \texttt{N} * [\texttt{P} = v_4] * [0 < \texttt{T}] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{R}, v_1, 2, [0,1]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad v_6 + 1 \mapsto v_8 * v_6 \mapsto nth(v_7, 0) * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(nth(v_7, 0), v_7, 2, [0]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{N}, v_3, 2, [0]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_6 :: v_7 :: v_8 :: nil, v_6), 2)\texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_7). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_6 :: v_7 :: v_8 :: nil, v_0), 2) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_3). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_3, v_0), 2)\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{T}\ \texttt{inTree}\ v_1]) * [\texttt{N} = v_5])) * \\
\quad\quad\quad\quad\quad [\texttt{tmp\_l} = 0] * [\texttt{tmp\_r} = 0] * [0 > v_6]) * - \\
\quad\quad (\texttt{AbsExists}\ v_1(\texttt{AbsExists}\ v_0 \\
\quad\quad\ v_6 + 1 \mapsto v_2 * v_6 \mapsto v_1)) * \\
\quad\quad [\texttt{I} = \texttt{tmp\_l}])))))))))
\end{array}
$$

PEDANTIC has a rewrite rule of the form $find(x, [x, ...]) \rightarrow [x, ...]$. Applying this rule to the fourth line of the above yields $nth(v_6 :: v_7 :: v_8 :: nil, 2)$. Evaluating the nth function yields the separation logic expression in State 4.4.2.

State 4.4.2

$$
\begin{array}{l}
(\texttt{AbsExists}\ v_8\ (\texttt{AbsExists}\ v_7 \\
\quad (\texttt{AbsExists}\ v_6\ (\texttt{AbsExists}\ v_5(\texttt{AbsExists}\ v_4 \\
\quad\quad (\texttt{AbsExists}\ v_3\ (\texttt{AbsExists}\ v_2(\texttt{AbsExists}\ v_1 \\
\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ v_6 \\
\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{T}\ (v_6+1) \\
\quad\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_r}\ (\texttt{T}+1) \\
\quad\quad\quad\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ \texttt{T} \\
\quad\quad\quad\quad\quad\quad\quad (\texttt{P}+1 \mapsto \texttt{T} * \texttt{P} \mapsto \texttt{N} * [\texttt{P} = v_4] * [0 < \texttt{T}] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{R}, v_1, 2, [0,1]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad v_6+1 \mapsto v_8 * v_6 \mapsto nth(v_7, 0) * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(nth(v_7,0), v_7, 2, [0]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{N}, v_3, 2, [0]) * \\
\quad\quad\quad\quad\quad\quad\quad\quad [v_8\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_7). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_6 :: v_7 :: v_8 :: nil, v_0), 2) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_3). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_3, v_0), 2)\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad\quad\quad\quad [\texttt{T}\ \texttt{inTree}\ v_1]) * [\texttt{N} = v_5])) * \\
\quad\quad\quad\quad\quad\quad [\texttt{tmp\_l} = 0] * [\texttt{tmp\_r} = 0] * [0 > v_6]) * - \\
\quad\quad (\texttt{AbsExists}\ v_1(\texttt{AbsExists}\ v_0 \\
\quad\quad\ v_6+1 \mapsto v_2 * v_6 \mapsto v_1)) * \\
\quad\quad [\texttt{I} = \texttt{tmp\_l}]))))))))
\end{array}
$$

The final tricky step is to reduce $find(v_6 :: v_7 :: v_8 :: nil, v_0)$ to $find(v_7, v_0)$. The trick is to recognize that v_0 will never be v_6. Several pieces of information from the context need to be used to verify this fact. We need to combine $v_0 \in \texttt{TreeRecords}(v_7)$ with $\texttt{TREE}(nth(v_7, 0), v_7, 2, [0])$ to verify that v_0 is one of the records in that tree. We then need to combine this with the $v_6 \mapsto \texttt{nth}(v_7, 0)$ predicate to verify that any element in the tree is distinct from v_6. The final result is State 4.4.3.

State 4.4.3

$$
\begin{array}{l}
(\texttt{AbsExists}\ v_8\ (\texttt{AbsExists}\ v_7 \\
\quad (\texttt{AbsExists}\ v_6\ (\texttt{AbsExists}\ v_5(\texttt{AbsExists}\ v_4 \\
\quad\ (\texttt{AbsExists}\ v_3\ (\texttt{AbsExists}\ v_2(\texttt{AbsExists}\ v_1 \\
\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ v_6 \\
\quad\quad\ (\texttt{AbsUpdateWithLoc}\ \texttt{T}\ (v_6+1) \\
\quad\quad\quad (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_r}\ (\texttt{T}+1) \\
\quad\quad\quad\ (\texttt{AbsUpdateWithLoc}\ \texttt{tmp\_l}\ \texttt{T} \\
\quad\quad\quad\quad (\texttt{P}+1 \mapsto \texttt{T} * \texttt{P} \mapsto \texttt{N} * [\texttt{P} = v_4] * [0 < \texttt{T}] * \\
\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{R}, v_1, 2, [0,1]) * \\
\quad\quad\quad\quad\quad v_6 + 1 \mapsto v_8 * v_6 \mapsto nth(v_7, 0) * \\
\quad\quad\quad\quad\quad \texttt{TREE}(nth(v_7,0), v_7, 2, [0]) * \\
\quad\quad\quad\quad\quad \texttt{TREE}(\texttt{N}, v_3, 2, [0]) * \\
\quad\quad\quad\quad\quad [v_8\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_7). \\
\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_7, v_0), 2) \\
\quad\quad\quad\quad\quad\quad\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad \texttt{AbsAll}\ v_0 \in \texttt{TreeRecords}(v_3). \\
\quad\quad\quad\quad\quad\quad [\texttt{nth}(\texttt{find}(v_3, v_0), 2)\ \texttt{inTree}\ v_1] * \\
\quad\quad\quad\quad\quad [\texttt{T}\ \texttt{inTree}\ v_1]) * [\texttt{N} = v_5)])) * \\
\quad\quad\quad\quad [\texttt{tmp\_l} = 0] * [\texttt{tmp\_r} = 0] * [0 > v_6]) * - \\
\quad\quad (\texttt{AbsExists}\ v_1(\texttt{AbsExists}\ v_0 \\
\quad\quad\ v_6 + 1 \mapsto v_2 * v_6 \mapsto v_1)) * \\
\quad\quad [\texttt{I} = \texttt{tmp\_l}]))))))))
\end{array}
$$

Now, the ALL phrase looks very much like the one we had in State 2.4.7 except that v_2 has been replaced with v_7. This allows other operations to fire that match this unfolded tree to the original.

PEDANTIC uses an inner rewrite algorithm. Many of the rewrites are implemented as mini-algorithms. All reductions reduce terms with respect to recursive path orderings. One of our goals is to reimplement the PEDANTIC separation logic simplifications (currently written in Gallina) using in our advanced rewriting library.

```
Coq < printAST 100.
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
(App (Construct (Name nat) 2)
...
(App (Construct (Name nat) 2)
(Construct (Name nat)
1))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
 f))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
Coq < Check 1.
1
     : nat

Coq < Check 1000.
1000
     : nat

Coq < Check 10000.
Warning: Stack overflow or segmentation fault happens when working
with large
numbers in nat (observed threshold may vary from 5000 to 70000
depending on
your system limits and on the command executed).
10000
     : nat

Coq <
```

Figure 4.5: Coq's failure to handle numbers larger than about 33000

4.4.3 Improve Coq's performance

In the process of integrating our rewriting system with Coq, we found that many of our optimizations are not implemented in Coq. Introducing these optimizations will likely both improve the speed and reduce the memory footprint of Coq.

4.4.3.1 Binary representation of integers

The Coq theorem prover uses a unary representation for natural numbers[4] (integers are built up from the constructors 0 and `S`. Figure 4.5 shows a Coq session illustrating some of the issues. Large integers will use up large amounts of memory. In fact Coq will generate a stack overflow for any integer larger than about 33000. A binary representation will substantially improve performance.

4.4.3.2 Interning of symbols and subterms

Interning of symbols will produce some performance improvement. However, interning of subexpressions will provide a substantial improvement. The interning of subterms and implementation of caching led to a several fold increase in the rewriting library when implemented years ago. We suspect there will be similar results when implemented in Coq.

[4]This issue will be fixed in the Coq 8.8 release

4.4.4 Improvements to our representation

The Coq theorem prover uses de Bruijn numbers to encode quantified variables. This is not done in our rewriting framework. Localized subexpression contexts are used to convert these indices to variable names. We suspect incorporating de Bruijn indices could improve the performance of our system.

4.4.5 Relationship to SMT solving

It is interesting to compare the algorithms in our library to those of an SMT solver. A C implementation of our library was developed (after the ML version) and became the basis of the Heuristic Theorem Prover [73]. Researchers have often observed that SMT solvers are substantially faster than Coq in verifying theorems. We suspect one of the issues may be the use of interned subexpressions in most SMT solvers.

4.5 Related research

Term rewriting theorem provers/program derivation systems

Many of the rewriting concepts used here come from work done in the Focus program derivation/theorem proving environment [70, 71]. The Focus system at its heart had a term rewriting system. Their system was extended to rewrite expressions with quantifiers. Some of our work on contextual simplification originated from work

done in the Focus system. We also based our conditional rewriting system on the conditional rewriting available in Focus. However, Focus never defined exact semantics or path orderings for its quantifiers and many of its rewriting strategies were not fully worked out. The Focus system in turn was built upon program transformation systems developed by Burstall and Darlington [24, 30].

The Isabelle simplifier

Isabelle is recognized for having a fairly powerful and extensible simplification system. However, our simplifier provides a number of advantages over Isabelle's simplifier. First, recursive path orderings ensure termination. The isabelle [62] manual shows an example where introducing the equation $f(x) = g(f(g(x)))$ could cause the simplifier to loop infinitely. In our system, recursive path orderings will translate this equation into $g(f(g(x))) \rightarrow f(x)$. The rule cannot be used the other way. Recursive path orderings will similarly disallow rules of the form $x + y = y + x$.

In fact, our handling of AC operators does not require any associative or commutative rules to be entered. The idea in our system is that an expression such as $a+(b+c)$ are flattened into $+(a, b, c)$. The rule matcher treats the arguments as a set rather than an ordered list. Hence, no rules are needed to switch around arguments to the AC operators. There is a similar mechanism for commutative operators such as equality. The matching algorithm for rewrite rules will automatically reorder $a = b$ to $b = a$ if necessary. Isabelle introduces a concept of ordered rewriting to partially

address AC matching. However, this strategy may require the user to choose an ordering and still is based on the idea of trying to get terms in the right order. Our system is fully automatic.

Isabelle provides a mechanism for introducing tactics to control simplification and introduce case analysis. We have not explored this area and in any case, one could use another mechanism such as LTac in Coq for this purpose. Isabelle also allows tactics to automatically split goals into cases or repeatly apply specialized simplification tactics as part of the simplification process. Again, this can be done using LTac in Coq.

Other Coq ML plugins

Braibant [22] developed a Coq plugin for rewriting modulo AC operators. His system uses type classes to identify AC operators. His system does not provide for the more algorithmic inner rewriting or many of the built in algorithms of our system. Their system automatically validates the AC rewrites with respect to the Coq meta-theory. Our system just produces a result; the proof term required to justify the rewrite with respect to Coq's logic is not produced. Instead a second goal is generated that the user needs to manually prove to validate the result. Braibant's plugin has support for neutral terms. It turns out that in our system that most of the benefits of neutral terms can be obtained by introducing rules such as $x + 0 \to x$.

SMTCoq [41] integrates SMT solving capabilities into Coq. SMTCoq converts

a formula in Coq to an SMT solver format. Once an unsat result is produced, the explanation is extracted and converted into a proof using Coq's meta theory. This integration allows formulae to be proven. However, there is no concept of simplification if the expression cannot be completely proven.

4.6 Conclusion and future work

One of the more interesting results of the work in this chapter is identifying opportunities to improve the performance of the Coq kernel. However, in addition to finding optimizations in our system that can be integrated into Coq, we also found optimizations in Coq that could improve the performance of our system.

There are a number of continuing projects we are planning. First, we still need to complete our interface to Coq from section 4.3. Second, we need plan to integrate this rewriting library into our PEDANTIC system. We may well need to add some customized code for some of the constructs in PEDANTIC. This will test the library on larger problems. Third, we still need to do work to generate justifications of our rewrites with respect to the Coq meta theory. We could also integrate optimizations found in Coq into our system. However, this last item is lower priority.

There is also a C implementation of the library which the author may make public at a future time. Development of this library was motivated by the fact that interning and decoding subexpressions as described in Section 4.2.2.2 can be done more cleanly

in C. Expressions would be stored in a giant DAG. The intern numbers are simply the memory addresses for the corresponding records. Pattern matching is done directly on this DAG. One does not need the `intern_exp` and `decode_exp` functions sprinkled throughout the code.

4.6.1 Coq Interface

Much of the work interfacing the rewriting library to Coq is complete. However, the interface is still buggy and a few key features still need to be completed. Most notably, the ability to introduce directives is incomplete. We describe the types of directives that are needed.

4.6.1.1 Declaration of precedence

In addition to symbol precedences that are automatically inferred from the environment (described in section 4.3.3.1), the user will be able to enter symbol precedence orderings through a `Precedence` command.

4.6.1.2 Declaration of rewrite rules

In addition to the generated rules, it may be useful for the user to add rewrite rules based on lemmas or that are missed by the automatic rule generation. This will be implemented through the introduction of forceRewriteRule and RewriteRule properties. The latter will introduce the rules even if they violate recursive path

constraints. There is a risk that using the latter will cause the simplifier to go into an infinite loop.

4.6.1.3 Declaring operator properties

The user will be able to add type class definitions to identify AC, equality, total order and partial order operators.

4.6.2 More closely modeling Coq's logic

The advanced rewriting library was written many years ago without the intention of connecting it to Coq or any other theorem prover. In fact, the intent was to develop a standalone theorem prover based on rewriting principles. The issue that arises is that many of the constructs in Coq do not have exact correspondences in our library. Two notable issues are 1) there is no support for dependent types and 2) There is no distinction between the uncomputable `Prop` and computable boolean expressions. Our short term solution is to use type inferencing algorithms to figure out the intended mappings when converting expressions back to Coq. Our longer term solution is to make changes to the advanced rewriting library to more closely follow Coq's logic.

4.6.3 Rewriting Soundness

In order to verify soundness of rewrites with respect to Coq's logic, we will integrate code which constructs a Coq correctness proof along with the rewrite generated. As our rewriter is a combination of many smaller algorithms, our soundness strategy will involve generating a proof of correctness for each rewrite. To start, our rewriter will have an algorithm which breaks this top level goal into subgoals to verify the correctness of each individual step. Each algorithm will then be responsible for producing a proof to verify the result that is produced. This will then be combined into the tree to produce the final proof.

Chapter 5

The DPLL verification

In this chapter we describe our work towards verifying the integrity of the data structure invariant for DPLL. Note that we are not doing a complete correctness verification, we are only verifying the integrity of the data structures.

Our DPLL algorithm implements a portion of the algorithm described in [77]. The algorithm implements the two watch variable data structure but not any of the machine learning. The C code is shown in Appendix D. We translated this code to our PEDANTIC data structure for imperative programs. This is shown in Appendix E. The top level verification proof is shown in Appendix G. This top level proof depends on 82 lemmas. These lemmas include such things as verifying preconditions for an operation (such as the range of an array reference), state entailments (at the end of a while block or elsewhere) and merges at the end of if-then-else constructs. Of these 82 lemmas, we have completed 8. There is one additional lemma which is a fairly

complex merge much of which is complete (`mergeTheorem2`). We describe this lemma in more detail later on in the Chapter. We anticipate that there will be a few other lemmas of similar complexity and that all the other lemmas while not completely trivial will be much simpler.

Earlier SAT solver verifications involved verifying a functional program such as in [63]. There have also been many efforts to create tools that verify the results produced by a SAT solver such as [11]. Our verification is based in an imperative implementation of DPLL. While we only verify the integrity of the data structure, the verification could be augmented to verify the correctness of the the output.

5.1 How the C code works

The SAT solver algorithm finds assignments for a boolean logic expression in conjunctive normal form. The starting point of the SAT solver algorithm is an algorithm that systematically tries all possible assignments for boolean variables. Variables rather than being given names are given numbers between 0 and `VAR_COUNT-1`. The system finds the first unassigned variable. It first tries making the variable `False` (represented by 1). If a contradiction is found, then the value is undone and the variable is set to `True` (represented by 2). If this fails, then the variable is unassigned and if the previous assignment was `False` (and not generated by a unit propagation, described below), it is changed to `True`. If the previous assignment was also `True`,

then this previous assignment is backtracked out as well until we find a variable that can be flipped.

The above algorithm is augmented with a unit propagation algorithm. After a variable is assigned, the system does a search for all variables whose values can be deduced. To do this, we first observe that for each clause, there must be at least one satisfying assignment. If all variables but one are assigned a value and those values do not satisfy the clause, then the last variable can be deduced. The DPLL algorithm employs a system of watch variables in which the system marks two unassigned variables in each clause as watched. After an assignment is done, the system traverses all clauses in which the particular variable is watched. If the assignment does not satisfy the clause and if no other assignment satisfies the clause, then the system first tries to find another unassigned variable to watch. If that cannot be done, then the other watch variable is the one and only unassigned variable in the clause and its assignment can be deduced. More specifically, the following three cases exist as our watch variable invariant:

- The two watch variables are unassigned variables. This is the normal state.
- All but one variable is assigned in a clause. One watch variable is the one unassigned variable. The other watch variable is the most recent assignment affecting the clause.
- At least one assignment already satisfies the clause. The watch variables might

be variables assigned values. However, it must always be the case that any watch variable that is assigned was either a satisfying assignment or happened after a satisfying assignment for the clause.

We now examine how these cases work for the following sample clause:

$$A \vee \neg \mathbf{B} \vee \mathbf{C} \vee \neg D$$

Initially, any two variables can be picked as watches. In the above clause the variables are B and C have been chosen arbitrarily and are marked in red. Now consider what happens when we start making assignments. We will start with $A = \texttt{False}$. The assigned variable is boxed. We do not need to change the watch variables.

$$\boxed{A} \vee \neg \mathbf{B} \vee \mathbf{C} \vee \neg D$$

Now, lets add an assignment for B so that we have $A = \texttt{False}, B = \texttt{True}$. B is no longer allowed to be a watch variable since it is assigned. We replace the watch variable B with D.

$$\boxed{A \vee \neg B} \vee \mathbf{C} \vee \neg \mathbf{D}$$

Next we add an assignment for C giving $A = \texttt{False}, B = \texttt{True}, C = \texttt{False}$. Now, we would like to find another variable to replace C as the watch variable. This cannot be done. In this case the watch variable does not move and we are in case 2 of the

invariant above.

$$\boxed{A \vee \neg B \vee \mathbf{C}} \vee \neg\ \mathbf{D}$$

Let's go back to where the only assignment was $A = \texttt{False}$,

$$\boxed{A} \vee \neg\mathbf{B} \vee \mathbf{C} \vee \neg D$$

if we were to add $B = \texttt{False}$, then we would have the following:

$$\boxed{A \vee \neg\mathbf{B}} \vee \mathbf{C} \vee \neg D$$

Since $B = \texttt{False}$ satisfies the clause, we move to case 3 of the watch variable invariant and do not need to change the watch variables. We could also add an assignment for D giving $D = \texttt{True}, B = \texttt{False}, A = \texttt{False}$ and we still would be in case 3 with the state as shown below.

$$\boxed{A \vee \neg\mathbf{B}} \vee \mathbf{C} \vee \boxed{\neg D}$$

5.1.1 Data structures

The program builds five data structures in heap memory to store the clauses of the formula being solved, the current assignments, the watch variables and a queue of unit propagations that need to be processed. A picture of these data structures is

shown in figure 5.1. This instance represents the formula $(A \vee \neg B \vee C) \wedge (A \vee \neg C \vee D)$. The variable A is assigned the value `True`. The other three variables are unassigned. For the first clause, the variables B and C are the two watch variables. For the second clause, C and D are watched. The declarations of these data structures can be found in Appendix D at positions marked ℓ_{D1} through ℓ_{D5}. We now describe the meaning of these five data structures.

- The variable `assignments` (at ℓ_{D1} in Apendix D) contains the variable assignments (1=false, 2 = true, 0=unassigned). Variables are numbered 0 through `VAR_COUNT-1`.

- The variable `clauses` (at ℓ_{D2} in Apendix D) contains a pointer to a linked list of the clauses. Each record contains one conjunct. The arrays `positive_lit` and `negative_lit` specify which variables occur positively and negatively respectively in the clause. The array `watch_var` specifies which two of the variables are being watched. Each element is a boolean specifying whether the variable corresponding to the index is being watched. `watch_next` and `watch_prev` are used to form a linked list of watch variables for each variable.

- For each variable, a linked list of all the clauses in which that variable is being watched is maintained. The variable `watches` (at ℓ_{D3} in Apendix D) contains the head of the watch linked list for each variable. Of the `VAR_COUNT` locations in the array, only the two corresponding to the watch variables of the clause are

actually used. The `watch_next` array contains pointers to the next clause for each watched linked list. The `watch_prev` array is used to maintain back links in order to make deleting an element in the middle efficient.

- The `assignments_to_do_head` and `assignments_to_do_tail` (both at ℓ_{D4} in Apendix D) variables point to the head and tail of a doubly linked list which maintains a queue of assignments that need to be completed. A single assignment may cause many other assignments to be deduced. The queue saves the deduced assignments so they can be processed one at a time.

- The `assignment_stack` (at ℓ_{D5} in Appendix D) linked list stores information on variable assignments in the order that the assignments were made. This makes backtracking and undoing assignments more straight forward. The `unit_prop` field stores whether or not the assignment came from a unit propagation. This information is used to decide where to stop when backtracking. If an assignment being undone resulted from a unit propagation, then it cannot be flipped. We need to continue backtracking.

5.1.2 Walk through of code

We now discuss the functionality of the code in Appendix D. The algorithm is implemented by the function `solve()` at ℓ_{D6}. The main loop of the algorithm starts at ℓ_{D8}. At this point, we need to do one of two things (depending on the value of

`backtrack`). Either a fresh unassigned variable needs to be picked for assignment or if we were backtracking, we need to flip a variable. When doing a fresh assign, we always initially assign the value `False` (1). When flipping the assignment, it is always from `False` (1) to `True` (2).

At ℓ_{D9}, we first check if a variable has been found for assignment. If not, then all variables have been assigned and we found a satisfying assignment. This case will never happen if we were backtracking at ℓ_{D8}. Assuming we have a variable, the next step is to assign it a value. This assignment will always be `False` (1) if a newly found variable is assigned or `True` if we were previously backtracking. Rather than directly making the assignment, we add the assignment to the todo list created with `assignments_to_do_head` and `assignments_to_do_tail`.

Next, at ℓ_{D10}, we have an inner loop which processes all the assigns on the todo list. We start with the one assignment from above. Processing an assignment may introduce zero or more new assignments through unit propagations. Each iteration of the loop processes one unit propagation. The loop will end when either 1) there are no more assignments to process, or 2) a contradiction is found and we need to backtrack. These cases correspond to the two cases at the beginning of the outer loop at ℓ_{D8}.

At ℓ_{D10}, we start by removing the assignment at the head of the todo list. We next check if the assignment has already been made at ℓ_{D11}. If the assignment has been made (this is only the case when processing a unit propagation), we check if it

is consistent. If it is consistent with the new assignment we intended to add, we skip to the next loop iteration (or finish). If not, there is a contradiction, then we need to backtrack. The logic for these cases can be found at ℓ_{D17}.

Assuming the assignment has not been made, we add the assignment to the `assignments` array and `assignments_stack` data structures. Next, at ℓ_{D12}, there is a loop that walks through all of the clauses watching the variable that has been assigned (or flipped). For each of these clauses, either there is a satisfying assignment, or the watch variable needs to be moved to another variable that is not assigned. If it cannot be moved, then there will only be one unassigned variable left and a new unit propagation needs to be added to the todo list maintained by the `assignments_to_to_head` and `assignments_to_do_tail` lists.

At ℓ_{D13} we check whether the new assignment itself satisfies the clause. At ℓ_{D14}, we have a loop that finds an unassigned variable. We also check to see if any other variable satisfies the clause in this loop. If so, we skip to the next clause. At ℓ_{D15} is code that will first add the new watch variable and then remove the existing watch variable if a candidate was found. At ℓ_{D16} is code that will first loop through to find the one remaining unassigned variable in a clause and then add a unit propagation to the todo list maintained by `assignments_to_to_head` and `assignments_to_do_tail`.

The code at at ℓ_{D17} is reached when the variable we intended to assign has already been assigned. If the assignment we intended is a contradiction, then we backtrack. The todo list is completely cleaned out. The variable stack is popped and assignments

are removed until a `False` assignment is found which did not result from a unit propagation. This assignment is flipped to `True` and we return to the next iteration of the outer loop. If no variable is found that can be flipped, then the program returns with an unsat result.

5.2 The DPLL invariant

Appendix F shows our invariant. In this section, we walk through the individual pieces of the invariant. The version of the invariant we trace is the loop invariant that exists at the beginning of the outer most loop of our DPLL algorithm (ℓ_{D8} in Appendix D). There are slight variations as we walk through the code and get to the inner loops.

The invariant starts with `coreStructures` at ℓ_{F5} (in Appendix F) which declares the space used by the five data structures from the previous section. Its declaration Appendix F at ℓ_{F1}. There are three `TREE` declarations for the `clause` linked list, the `assignments_to_do` linked list and the `assignment_stack` linked list. Then there are two arrays for the `assignments` array and the `watches` array of pointers for the heads of the watch variable linked lists.

Now we start stating relationships between the data structures. We first verify the integrity of the assignment data structures, `assignments` and `assignment_stack` (ℓ_{D1} and ℓ_{D5} in Appendix D). This is done by the predicate `treeEquivArray` at

ℓ_{F6} (in Appendix F). It is defined at ℓ_{F4}. This declaration breaks down into two parts. The first verifies that each record in the `assignment_stack` corresponds to a non-zero assignment in the `assignments` array. It also verifies that no element is duplicated in the stack, that the corresponding positions are either 1 (for negative) or 2 (for positive) and that all offsets into the array are smaller than `VAR_COUNT`. This is shown at ℓ_{F2}. The second part verifies that any non-zero location in the `assignments` array corresponds to a record in the `assignments_stack` structure. The `arrayInTree` declaration appears at ℓ_{F3}.

Next we have an assertion to verify the integrity of the `assignments_to_do_head` and `assignments_to_do_tail` lists (ℓ_{F7} in Appendix F). Since this invariant is at the beginning of the outer loop (ℓ_{D8} in Appendix D), we have an assertion that this list is empty at ℓ_{F7} (in Appendix F). However, after this at ℓ_{F8} (in Appendix F), there is a `validBackPointers` assertion (commented out). This ensures that the the `prev_offset` is in fact a pointer to the previous record. It is commented out as this list is always empty. However, within the loop, inner loops populate the list so we show the `validBackPointers` assertion for reference.

5.2.1 The watch variable invariants

At the beginning of the outer loop, there are two possibilities for the watch variables. If the variable `backtrack` is false, then we are ready to select a new variable for assignment. In this case, all clauses have at least two unassigned variables. In the

second case, where `backtrack` is true, the most recent variable assignment has been flipped. In this case, clauses have either one or two unassigned variables. However, if there is only one unassigned, then the most recent variable assignment removed one of the unassigned variables. This is asserted at ℓ_{F9} (in Appendix F). Note this invariant is in addition to the basic watch variable invariant cases discussed in Section 5.1.

The next clause ℓ_{F10} (in Appendix F) asserts that if we undo all the unit propagations that all clauses will have at least two unassigned variables.

Now we have assertions to verify the integrity of the watch variable linked lists. There is a linked list for each of the variables `0` through `VAR_COUNT-1`. These linked lists do not allocate any new records but are built up within the records already allocated for the clauses. ℓ_{F11} has a quantifier to enumerate over the variables. ℓ_{F12} introduces a `Path` construct. This is similar to `TREE` but does not introduce any newly allocated memory. It specifies structures built inside of an existing structure. Finally, at ℓ_{F13} is the assertion verifying integrity of the back pointers for the watch variable linked lists.

The assertion at ℓ_{F14} states that given the current assignments, either the clause is already satisfied or that there is at least one unassigned variable.

At ℓ_{F15} is an assertion stating that any non-zero element in the `watch_var` array corresponds to an actual literal in the clause (either the corresponding position in `positive_lit` or `negative_lit` is non-zero). The assertion at ℓ_{F16} states that the clause is on the watch linked list for a particular variable if and only if the corre-

sponding element in the `watch_var` array is non-zero. ℓ_{F17} is the assertion that every clause has two watch variables.

Finally, a clause can be in one of three possible states with respect to its watch variables (as described in section 5.1):

- The two watch variables are unassigned variables (ℓ_{F21} in Appendix F). This is the normal state.
- All but one variable is assigned in a clause. One watch variable is the one unassigned variable. The other watch variable is the most recent assignment affecting the clause (ℓ_{F18} in Appendix F).
- At least one assignment already satisfies the clause. The watch variables might be variables assigned values. However, it must always be the case that any watch variable that is assigned was either a satisfying assignment or happened after a satisfying assignment for the clause (ℓ_{F19} and ℓ_{F20} in Appendix F).

5.3 Overview of the DPLL verification

We now discuss how the DPLL proof is being developed. Appendix G shows the top level proof of our DPLL implementation. Appendix E shows the PEDANTIC representation of our DPLL program. The C program is converted to the Coq data structure by hand. We refer to these two figures as describe the verification.

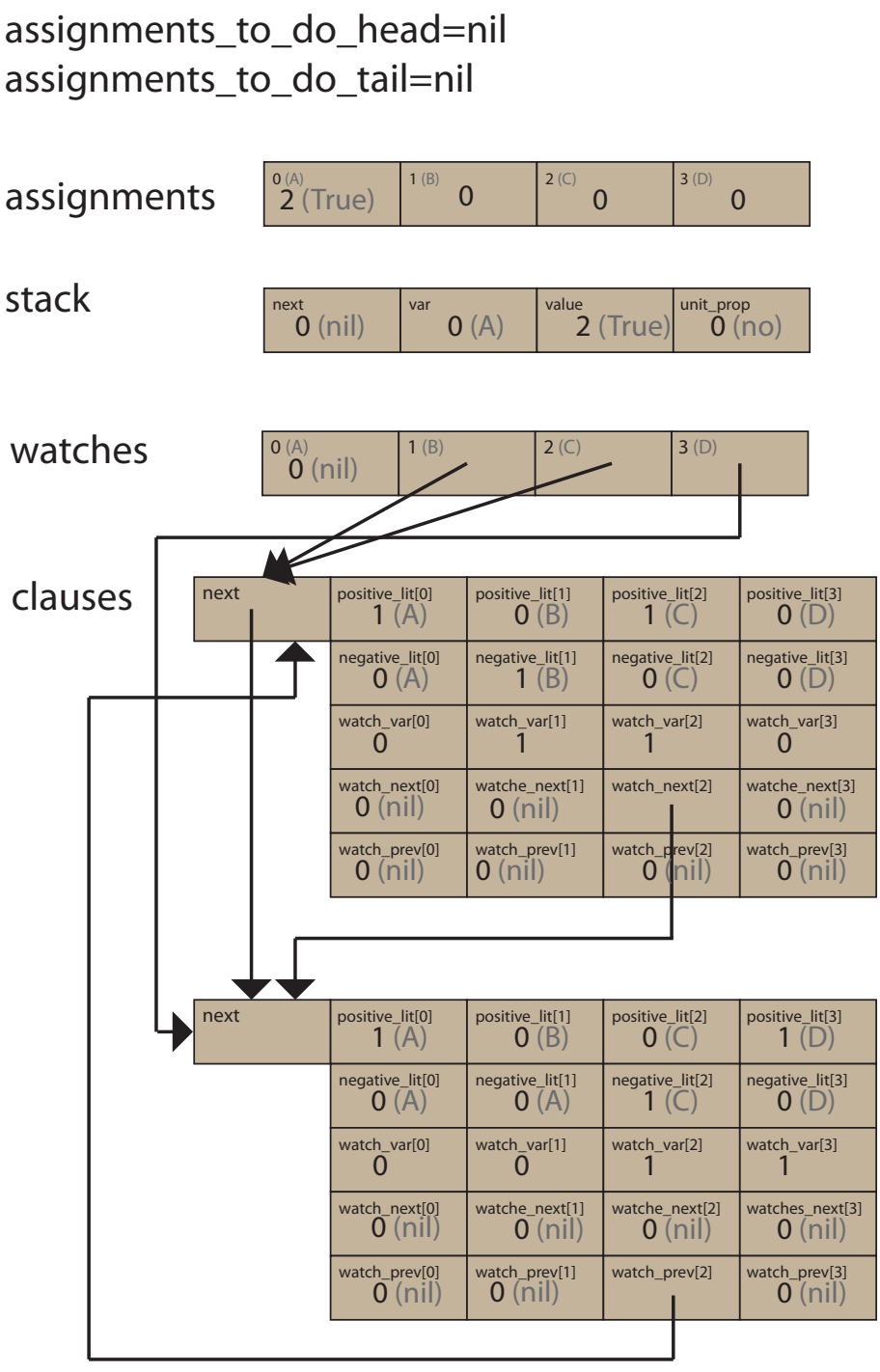


Figure 5.1: Picture of the DPLL data structure

```
mergeStates
    (AbsUpdateLoc
       (AbsUpdateVar
          (AbsUpdateVar
             (AbsMagicWand
                (AbsUpdateWithLoc
                   (AbsUpdateVar
                      (AbsUpdateWithLoc
                         (AbsUpdateVar
                            ([!! (backtrack)] **
                             AbsUpdateVar ([# 1] ** loopInvariant)
                               have_var # 0) backtrack
                            # 0) varx !! (stack) ++++
                                 # stack_var_offset)
                         valuex # 2) ssss !! (stack) ++++
                              # next_offset)
                (AbsExistsT
                   (AbsExistsT
                      (AbsExistsT
                         (AbsExistsT
                            (AbsExistsT
                               (v( 0) ++++ # 3 |-> v( 4) **
                                v( 0) ++++ # 2 |-> v( 3) **
                                v( 0) ++++ # 1 |-> v( 2) **
                                v( 0) ++++ # 0 |-> v( 1) **
                                [!! (stack) ==== v( 0)])))))))
             stack !! (ssss)) have_var # 1)
                  !! (assignments) ++++ !! (varx) # 0)
    ([~~ (convertToAbsExp (! iiii <<= ANum var_count))] **
           haveVarInvariant)
    invariant.
```

Figure 5.2: mergeTheorem2 statement

```
mergeStates
    (AbsExistsT
 (AbsExistsT
  (AbsExistsT
   (AbsExistsT
    (AbsExistsT
     (AbsExistsT
      (AbsExistsT
       (AbsExistsT
        (AbsExistsT
         (AbsExistsT
          ([nth( v( 3), !! (varx)) ==== # 0] **
           ((((([!! (ssss) ==== nth( v( 7), # 0)] **
               (AbsEmpty **
                ((((((TREE( !! (clauses), v( 2), # 21, # 0 :: nil) **
                      TREE( nth( v( 7), # 0)
                          , v( 7), # 4, # 0 :: nil) **
                      ARRAY( !! (watches), # 4, v( 4))) **
                     AbsClosure treeEquivArray (list
                          ( v( 6) :: v( 7) :: !! (varx) :: v
                           ( 8) :: v( 9) :: nil) :: replacenth
                          ( v( 3), !! (varx), v( 0)) :: nil) **
                     ([!! (assignments_to_do_head) ==== # 0] **
                      [!! (assignments_to_do_tail) ==== # 0]) **
                     ([# 0 <<<< v( 5)] **
                      AbsClosure atleastTwoUnassignedOrLast
                          (v( 2) :: replacenth( v( 3), !!
                            (varx), v( 0)) :: list
                           ( v( 6) :: v( 7) :: !!
                            (varx) :: v( 8) :: v( 9) :: nil) :: nil)
                                *\/* AbsClosure atleastTwoUnassigned
                          (v( 2) :: replacenth( v( 3), !!
                            (varx), v( 0)) :: v( 5) :: v
                           ( 1) :: nil)) **
                     AbsClosure twoUnassignedAtChoiceSteps
                         (v( 2) :: replacenth( v( 3), !!
                           (varx), v( 0)) :: list( v
                           ( 6) :: v( 7) :: !! (varx) :: v
                           ( 8) :: v( 9) :: nil) :: nil) **
...
```

Figure 5.3: First part of left branch for merge after expanding out magic wand, `absUpdateVar` and `absUpdateWithLoc`

```
doMergeStates
    (AbsEmpty **
 ((((([!! (ssss) ==== nth( v( 7), # 0)] **
     (AbsEmpty **
      ((((((AbsEmpty ** AbsEmpty ** AbsEmpty) **
           AbsClosure treeEquivArray (list( v( 6) :: v
                 ( 7) :: !! (varx) :: v( 8) :: v( 9) :: nil) ::
                  replacenth
                ( v( 3), !! (varx), v( 0)) :: nil) **
           (AbsEmpty **
            [!! (assignments_to_do_tail) ==== # 0]) **
           ([# 0 <<<< v( 5)] **
            AbsClosure atleastTwoUnassignedOrLast
                (v( 2) :: replacenth( v( 3), !! (varx), v
                  ( 0)) :: list( v( 6) :: v( 7) :: !!
                  (varx) :: v( 8) :: v( 9) :: nil) :: nil) *\/*
                       AbsClosure atleastTwoUnassigned
                (v( 2) :: replacenth( v( 3), !! (varx), v
                  ( 0)) :: v( 5) :: v( 1) :: nil)) **
           AbsClosure twoUnassignedAtChoiceSteps (v
                ( 2) :: replacenth( v( 3), !! (varx), v
                 ( 0)) :: list( v( 6) :: v( 7) :: !!
                 (varx) :: v( 8) :: v( 9) :: nil) :: nil) **
...
([# 4 <<<< !! (iiii)] **
 (((AbsEmpty ** AbsEmpty ** AbsEmpty **
    AbsEmpty) **
   AbsClosure treeEquivArray (v( 1) :: v( 2) :: nil) **
   (AbsEmpty **
    AbsEmpty) **
   ([# 0 <<<< !! (backtrack)] **
    AbsClosure atleastTwoUnassignedOrLast (v( 0) :: v
         ( 2) :: v( 1) :: nil) *\/* AbsClosure
          atleastTwoUnassigned
        (v( 0) :: v( 2) :: !! (backtrack) ::
          !! (varx) :: nil)) **
   AbsClosure twoUnassignedAtChoiceSteps (v( 0) :: v
        ( 2) :: v( 1) :: nil) **
...
```

Figure 5.4: Portions of left and right hand sides after pairing off all identical pieces

```
AbsClosure treeEquivArray (list( v( 6) :: v
              ( 7) :: !! (varx) :: v( 8) :: v( 9) :: nil) ::
              replacenth
             ( v( 3), !! (varx), v( 0)) :: nil)

AbsClosure treeEquivArray
     (v( 7) :: v( 3) :: nil)) bbb (eee, empty_heap)
```

Figure 5.5: Left and right hand sides of `treeEquivArray`

We note that portions of this proof are quite complex. However, one should observe that when the proof development tools improve, doing these types of complex proofs will likely be considerably cheaper than shipping software with bugs that were not caught by QA.

In Appendix G, the first few steps (starting at ℓ_{G1}) performs some setup. The program definition is unfolded. We instantiate an existential variable. The next step (`strengthenPost`) is needed because our goal is a complete Hoare triple. When we forward propagate over the steps of the program, the generated state may not match the final goal. `strengthenPost` replaces that final state with an existential variable and setups an entailment proof (for later on) to verify that the generated state implies the post condition.

Next, there is an `apply compose` that decomposes two sequential statements. The `pcrunch` applies the Hoare rule for assignment for the statement before the while loop at ℓ_{E1}.

The first application of `whileThm` (at ℓ_{G3} in figure G) is used to introduce the invariant for the outer while statement at ℓ_{E1} in the program in Appendix E. Next

at ℓ_{G4}, we abstract the post condition and then the `pcrunch` propagates over many of the statements in the while loop. It automatically walks into both branches of the `IF-THEN-ELSE`. On the left branch it propagates over all statements. On the right branch, it will stop at the while statement at ℓ_{E6}. This propagation introduces proof obligations. `PreCond1` (ℓ_{G5}) is an auxiliary proof that verifies that the `DELETE` at ℓ_{E4} frees allocated memory. `PreCond2` (ℓ_{G6}) verifies that the `CStore` at ℓ_{E5} references valid memory. We then use `whileThm` (ℓ_{G7}) to introduce the invariant for the while statement at ℓ_{E6}. We then have the auxiliary theorem `mergeTheorem1` (ℓ_{G9}) that merges the two branches of the `IF` statement at ℓ_{E8}. The merge point is at ℓ_{E9}. The goal is shown in figure 5.6. Since only a couple of variables were changed in the then branch, this merge is fairly straight forward. The merged state simply deletes information about `varx` and `have_var`. We now have the theorems `entailment1` (ℓ_{G10}) and `entailment2` (ℓ_{G11}). The first validates that the state just before ℓ_{E10} implies the loop invariant. The second validates that the abstract state right before the while loop at ℓ_{E6} implies the invariant of the while loop. Finally, we have `mergeTheorem2` (ℓ_{G12}) which merges the `THEN` and `ELSE` branches at ℓ_{E11}. This is a fairly complex proof as one branch (but not the other) deletes a variable assignment. The challenge is that we need to verify that after deletion, our watch variable invariant still holds for each clause despite not moving around any of the watches.

Figure 5.2 shows the statement of the theorem for this merge. One should note that the left branch has many updates reflecting the removal of an assignment. There

```
mergeStates
    (AbsUpdateVar (AbsUpdateVar ([!! (ssss) ==== # 0] **
   AbsUpdateWithLoc ([~~ # var_count <<<< !! (iiii)] **
    haveVarInvariant) ssss !! (assignments) ++++ !!
       (iiii)) varx !! (iiii)) have_var # 1) ([~~ !!
     (ssss) ==== # 0] **
 AbsUpdateWithLoc ([~~ # var_count <<<< !! (iiii)] **
  haveVarInvariant) ssss !! (assignments) ++++ !!
     (iiii)) ?Q2
```

Figure 5.6: The goal state right before `mergeTheorem1`.

is a magic wand to delete the top most element of the assignment stack. There is also an `AbsUpdateLoc` that changes the appropriate location in the `assignments` array to 0 reflecting an unassigned variable.

The first step is to expand out all the occurences of `AbsUpdateLoc`, `AbsUpdateVar`, `AbsUpdateWithLoc` and the `AbsMagicWand`. The semantics of these constructs were all described in Chapter 2. In order to remove the `AbsMagicWand`, one needs to first unfold the the first record of the assignment `stack`. This is the record being removed by the `magicWand`. The first part of the left branch after all the expansions is shown in figure 5.3. We note that after all of the above transformations, the left hand side becomes several hundred lines.

The result is considerably more complex than the original. One should note the repeated occurences of `replacenth(v(3),!!(varx),v(0))`. `v(3)` is an existential variable that stores the functional representation of the assignment array. When pairing off constructs, the `v(3)` needs to be paired to the corresponding functional variable for the right hand side. In some cases, this `replacenth` function simplifies out

and the pairing can be done automatically. However, there are many cases where this is not the case and a non-trivial proof needs to be completed to pair the constraints such as the one shown in figure 5.5. The lower predicate is the one we want in the merged state. There is a tactic in our system that will allow this expression to be accepted in the final state if the information in the left hand side entails the predicate. This ends up being a fairly involved proof and is not detailed here.

The most interesting part of the merge proof involves matching up the watch variable cases from Section 5.1. The proof involves six cases built up from the three cases at the end of section 5.2.1.

1. All but one variable is assigned. The variable being removed is not present in the clause. It is still the case that all but one variable is assigned and that the other conditions hold.

2. All but one variable is assigned. The variable being removed is present in the clause. We have an auxiliary constraint saying that the most recent assignment affecting the clause is indeed the other watch variable. Hence, the clause moves to the case where there are two or more unassigned variables and the watches are unassigned variables.

3. A satisfying assignment has been made. The variable being removed is not the satisfying assignment. The result is that we still have a clause in which a satisfying assignment has been made and that all the watches are either unassigned

variables or after the satisfying assignment.

4. A satisfying assignment has been made. The variable being removed is the satisfying assignment and there is no longer a satisfying assignment. Since we required watch variables to either be unassigned or pointing to the variable that had a satisfying assignment or later, we can deduce that all watch variables point to unassigned variables.

5. A satisfying assignment has been made. The variable being removed is the satisfying assignment and there still is another satisfying assignment. Since there is another satisfying assignment and the logic of the proof is the same as for the case above where we removed a variable that is not a satisfying assignment

6. The two watch variables point to two unassigned variables. This will still be the case after removing one more assignment.

5.4 Productivity issues

Despite the many contributions of CoqPIE described in Chapter 3, developing the DPLL theorem is still quite tedious. `mergeTheorem2` has about 20 lemmas so far. Each lemma consists of as many as 300 steps and a proof statement that may be several hundred lines.

The first issue is that replaying theorems is still quite slow despite modularization. It is the experience of the authors that the performance of Coq is directly related to the number of steps in a theorem and the size of the goal at each step. With complex goal invariants, it is hard to make the goal statements smaller despite our modularization. We have been able to improve performance by breaking theorems down into smaller pieces. However, this means that many more theorem statements have to be edited when an error is detected. This makes replay quite tedious.

One of our observations is that 80% of the time in replaying theorems is taken up by the code that simplifies invariants. We are required to use `simpl` due to the modularization described in Section 5.4.1. Currently our algorithm that does this simplification is quite slow. We have an ML implementation of a more comprehensive version of this algorithm as described in Chapter 4. We are working to integrate this library but work is not complete.

In addition to slowing down Coq, if a theorem is several hundred lines, it may be difficult to read. We have experimented with different pretty printing algorithms in CoqPIE to make these expressions more readable. CoqPIE's difference highlighting also helps out sometimes.

5.4.1 Invariant modularization

To mitigate the productivity issues above, we introduced invariant modularization. The goal of invariant modularization is two fold. First, we want to make the invariants

more readable by capturing details in definitions. Second, by capturing the details, we make the invariants smaller which helps make the Coq theorem prover faster.

Our representation of invariants uses a deep model. All of the basic constructs such as `AbsStar` are functors in a Coq inductive data type. We do not have a mechanism for definitions in our deep model. What we do instead is we use Coq's definition capability to pack portions of the invariant.

One issue with the deep model is that Coq function evaluation (`simpl` and `compute`) attempts to unfold definitions. Also, it is often useful to alter parameters passed to these functions. Instantiating a bound variable may need to alter a parameter. One cannot write Gallina functions that can detect where a Coq definition is used to package a portion of an invariant.

Our solution is to introduce an `AbsClosure` construct. This construct takes two parameters. The first is a separation logic data structure. The second parameter is a list of expressions. This construct is evaluated by first evaluating the expressions and then substituting these expressions for bound variables in the first parameter. For example, `AbsClosure ([v(0)]**[v(1)====3]) (#0::#1::nil)` is simplified to `[#0]**[#1====3]`.

One can now write an expression such as

```
Definition invariantNoTail
    (* clauses assignments_to_do stack assignments watches*) :
         absState :=
    (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT
       (AbsClosure invariantCoreNoTail (v(0)::v(1)::v(2)::v(3)::v(4)::
           (!!clauses)::(!!assignments_to_do_head)::(!!stack)::
```

```
(!!assignments)::(!!watches)::(!!backtrack)::
(!!assignments_to_do_tail)::nil)))))))).
```

The `AbsClosure` prevents tactic functions from touching the `invariantCoreNoTail` application. Parameters to the `AbsClosure` can be substituted as necessary.

This mechanism has mixed results with Coq. In addition to packaging applications inside an `AbsClosure`, one needs to make them Opaque to prevent unfolding. Also, one needs to use `simpl` rather than `compute` to prevent introduction of `fix` constructs (from unfolding definitions). One of the benefits of modularization is to improve the performance of Coq. However, the `AbsClosure` construct seems to limit simplification to `simpl`. The faster `vm_compute` and `native_compute` cannot be used either because they ignore `Opaque` declarations.

5.5 Summary

The DPLL verification has been quite challenging due to Coq performance issues. The verification is currently incomplete. For the portion that is complete, there are several thousand lines of proof script code which took several months to develop.

Chapter 6

Conclusion

The primary goal of our research is to create a practical tool for documenting and verifying data structure invariants. Many software bugs can be traced to data structure invariant violations. We chose to work on verifying DPLL because of the small program size and complexity of the invariant. The relationships described by the DPLL invariant are typical of the data structure relationships in many software systems.

6.1 PEDANTIC recap

In Chapter 2, we introduced our separation logic framework. The framework uses a deep model which represents the invariants using a data structure rather than directly using the Coq theorem prover's functional language (Galina). This gives us the ability

to write functions that traverse the invariants and do custom transformations. In many cases, we found that the existing tactics in the Coq theorem prover did not do exactly the right thing. We have also augmented separation logic with a few special constructs needed for the DPLL invariant. Many of these constructs we anticipate may be used for other program verifications.

6.2 The challenge of scale

When programs and their invariants are small, Coq works reasonably well. However, as invariants get larger, productivity slows down substantially. We were unable to finish the DPLL proof due to many productivity issues. First, Coq slows down substantially and uses considerably more memory as theorems and invariants become larger. As much as possible, we worked to keep theorems as well as the size of the goal states small. Often we broke a large theorem into several lemmas. This was not always possible. Second, we found ourselves going through many iterations of updating theorem statements and then replaying and updating the proof steps. Third, we found that reading and identifying portions of an invariant that need to be worked on quite tedious.

6.2.1 Top-down development

When proving complex theorems, we found that a top down approach was more productive than a bottom up approach. In the top down approach, we proved a main theorem. Often we introduced lemmas for details we did not want to address immediately. We found that it was better not to try and prove the lemmas until the main proof was done. Often these lemmas changed and proving them to soon wasted a lot of effort.

6.2.2 Partial verifications

One of the interesting properties of PEDANTIC is that top-level proofs resemble the structure of the original source code. Because the forward propagation of the separation logic states is automated, we anticipate that much of this top level proof can be automated. Verification engineers will of the option of not completing all subproofs and do a "light weight" verification.

6.2.3 Proof development productivity tools

As part of the research presented in this thesis, we are developing two tools to address the productivity issues of Coq. The tools are directly motivated by the productivity issues we are seeing with developing the DPLL proof. We are developing these tools concurrently with the DPLL proof. Having a proof development task at

hand provides the best method for validating the usefulness of a tool. It also gives us a unique opportunity to iterate on the designs of these tools as we use them in developing the DPLL proof.

6.2.3.1 CoqPIE recap

To mitigate many of the issues in Section 6.2, we developed the CoqPIE tool. It pre-runs and caches all theorems. This avoids the need to rerun proofs when browsing theorems. We also have tools to facilitate replaying of theorems. There is a project navigator to help users move around a large project with multiple source files. There is difference highlighting to identify key pieces of an invariant to work on.

There are still several areas where CoqPIE needs to be improved. We often find it necessary to propagate theorem revisions through many lemmas. We are looking into a replay capability that can update multiple theorems. Also, we need to make the information for replay persistent over project rebuilds. We found ourselves rebuilding the project fairly often. CoqPIE is currently written with Python's tkinter as the basis for the UI. This package is turning out to have many short comings. We are looking into switching to either Javascript or the Mac UI.

6.2.3.2 Rewriting library recap

Of the different pieces of proof replaying that were slow, the tactics to simplify invariants turns out to be the slowest. We are in the process of developing an ML

module that will perform these rewrites. This module is described in Chapter 4. There is a general purpose mechanism to enter conditional rewrite rules to specify some simplifications. An addition, the module integrates many different specialized algorithms for simplifying expressions. Rewriting is guaranteed to terminate as all algorithms and the rewriting rules are required to simplify with respect to recursive path orderings.

Among the many algorithms, there are generalized tools to simplify expression with operators that have associative or commutative properties. There is a contextual rewriting algorithm which adds additional rules when simplifying one subterm in a context where other subterm predicates in the expression are known to be true or false. There is a transitive rewriting algorithms for detecting cycles of transitive operators such as integer less than. There are many algorithms specialized to commonly used operators. For example, there is an algorithm that can detect linear equations and solve them for a particular variable or to do the common algebraic simplifications of collecting like terms.

6.3 Work needed to finish DPLL

Our plan for continuing this research is to first complete integration of the rewrite library into PEDANTIC. This should improve theorem replay performance substantially. Many of the issues we ran into with simplification and modularizing state

expressions are related to our decision to use a deep model. As we are now creating ml-plugins for Coq, we can switch to a shallow model. ML code will be used in place of the Gallina functions for manipulating separation logic states. Quite a bit of Gallina code in PEDANTIC will need to be translated to ML. This will be a fairly major refactoring of PEDANTIC. We will also work on adding some of the enhancements to CoqPIE described in Section 6.2.3.1. Once these tasks are complete, we will hopefully have increased productivity substantially and completing the DPLL proof will not be too tedious.

6.4 Beyond DPLL

After DPLL, our plan is to look at many other pieces of code that use complex data structures. We will be looking at code that is in wide spread use. One example that we are considering is the verification of the Linux malloc function. In order for malloc to find and release blocks of memory efficiently, some very complex data structures are used to link together free memory (which may be fragmented). It is a requirement that these functions both be fast and memory efficient. This code is an ideal target for our verification methodologies.

After a few well known pieces of code are verified, it is our hope to work with some businesses to verify commercial code and hopefully, the prcoess will find bugs.

We are also looking into light weight variants of our verification methodology. As

proofs are done top-down, it may be useful to only to the top-level proofs and omit the tedious details.

6.5 Impact on Software Development

The most important measure of success for the project is measure the ratio of proof development time to software development time. Right now this ratio is above 100. If it could be brought down to less than 10, then formal methods will provide a powerful alternative to conventional software verification methodologies.

6.6 Downloading source files

The main proof described in Chapter 2 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 described in Chapter 5 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 (from Chapter 3) and PEDANTIC are found at `https://github.com/kendroe/CoqPIE`. They compile with Coq 8.5pl3.

Our rewriting library from Chapter 4 is compiled with OCaml 4.04.0 and connected to Coq 8.5pl2. The library is available at `https://github.com/kendroe/CoqRewriter`. The source tree has two Makefiles. There is one makefile that will

produce the plugin for Coq. There is another Makefile that will build the rewriting library as a standalone program (with a test suite).

Appendix A

Tree traversal in Coq

```
Definition initCode : com :=
    I ::= A0;
    T ::= !RR;
    P ::= A0.

Definition loop : com :=
    WHILE ALnot (!T === A0) DO
        N ::= !P;
        NEW P,ANum(2);
        (CStore ((!P)+++(ANum 1)) (!T));
        (CStore ((!P)+++(ANum 0)) (!N));
        (CLoad Tmp_l ((!T)+++ANum(0)));
        (CLoad Tmp_r ((!T)+++ANum(1)));
        (CIf (ALand (!Tmp_l === A0) (!Tmp_r === A0))
            (CIf (!I === A0)
                 (T ::= A0)
                 ( (CLoad T ((!I)+++ANum(1)));
                   (CLoad Tmp_l ((!I)+++ANum(0)));
                   DELETE !I,ANum(2);
                   I ::= (!Tmp_l)))

            (CIf (!Tmp_l === A0)
                 (CLoad T ((!T)+++ANum(1)))
                 (CIf (!Tmp_r === A0)
                      (CLoad T ((!T)+++ANum(0)))
```

```
                    (N ::= !I;
                     NEW I,ANum(2);
                     (CStore ((!I)+++ANum(0)) (!N));
                     (CLoad Tmp_l ((!T)+++(ANum 1)));
                     (CStore ((!I)+++ANum(1)) (!Tmp_l));
                     (CLoad T ((!T)+++(ANum 0)))))))
    LOOP.
```

Appendix B

Top level proof for tree traversal

```
Theorem loopInvariant :
 afterInitAssignsloopafterWhile 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.

ℓ_B1  (* N ::= !P; *)
 eapply compose.  pcrunch.
ℓ_B2  (* 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 *)
ℓB3     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.
```

Appendix C

Model.v demo for CoqPIE

```
(*************************************************************
 *
 * Model.v
 *
 * A Small Coq model
 *
 * Developed by Kenneth Roe
 * For more information, check out www.cs.jhu.edu/~roe
 *
 *************************************************************)

Require Export SfLib.
Require Export SfLibExtras.

Inductive Exp :=
  | Var : nat -> Exp
  | Const : nat -> Exp
  | Plus : Exp -> Exp -> Exp
  | Times : Exp -> Exp -> Exp
  (*| Ref : Exp -> Exp*)
  | Find : Exp -> Exp
  | Ite : Exp -> Exp -> Exp -> Exp.

Fixpoint findit a l :=
    match l with
```

```
    | nil => 0
    | (f::r) => if beq_nat a f then (S 0) else
                  match findit a r with
                  | 0 => 0
                  | (S n) => S (S n)
                  end
    end.

Fixpoint eval e l :=
         match e with
         | Const v => v
         | Var n => nth n l 0
         | Plus a b => (eval a l)+(eval b l)
         | Times a b => (eval a l)+(eval b l)
         (*| Ref a => nth (eval a l) l 0*)
         | Find a => findit (eval a l) l
         | Ite a b c => match (eval a l) with
                        | 0 => eval c l
                        | S _ => eval b l
                        end
         end.

Fixpoint pop e v :=
         match e with
         | Const v => Const v
         | Var 0 => Const v
         | Var (S n) => Var n
         | Plus a b => Plus (pop a v) (pop b v)
         | Times a b => Times (pop a v) (pop b v)
         (*| Ref a => Ref (pop a v)*)
         | Find e => Find (pop e v)
         | Ite a b c => Ite (pop a v) (pop b v) (pop c v)
         end.

Fixpoint noFind e :=
    match e with
    | Const _ => true
    | Var _ => true
    | Plus a b => if noFind a then noFind b else false
    | Times a b => if noFind a then noFind b else false
    (*| Ref e => noFind e*)
    | Find e => false
```

```
    | Ite a b c => if noFind a then
           (if noFind b then noFind c else false) else false
    end.

(*Fixpoint noRef e :=
    match e with
    | Const _ => true
    | Var _ => true
    | Plus a b => if noRef a then noRef b else false
    | Times a b => if noRef a then noRef b else false
    | Ref e => false
    | Find e => noRef e
    | Ite a b c => if noRef a then
                (if noRef b then noRef c else false) else false
    end.*)

Theorem popEquiv : forall e v l, (* noRef e=true -> *)
            noFind e=true -> eval e (v::l)=eval (pop e v) l.
Proof.
    intro e.
    induction e.

    simpl. unfold eval. destruct n. reflexivity. reflexivity.

    unfold pop. unfold eval. reflexivity.

    intros. unfold noFind in H. fold noFind in H.
    remember (noFind e1). destruct b.

    unfold pop. fold pop. unfold eval. fold eval.

    rewrite IHe1. rewrite IHe2.
    reflexivity. apply H. reflexivity. inversion H.


    intros. unfold noFind in H. fold noFind in H.
    remember (noFind e1). destruct b.

    unfold pop. fold pop. unfold eval. fold eval.

    rewrite IHe1. rewrite IHe2.
    reflexivity. apply H. reflexivity. inversion H.
```

```
    intros. unfold noFind in H. inversion H.

    intros. unfold noFind in H. fold noFind in H.
    remember (noFind e1). destruct b.
    destruct (noFind e2).

    unfold pop. fold pop. unfold eval. fold eval.

    rewrite IHe1. rewrite IHe2. rewrite IHe3. reflexivity.

    apply H. reflexivity. reflexivity. inversion H. inversion H.
Qed.

Inductive Pr :=
          | Atom : Exp -> Pr
          | And : Pr -> Pr -> Pr
          | Or : Pr -> Pr -> Pr
          | Implies : Pr -> Pr -> Pr.

Fixpoint valid p l :=
    match p with
    | Atom e => if beq_nat (eval e l) 0 then false else true
    | And a b => if valid a l then valid b l else false
    | Or a b => if valid a l then true else valid b l
    | Implies a b => if valid a l then valid b l else true
    end.

Theorem validTransitive : forall a b c, (a -> b) ->
                (b -> c) -> (a ->c).
Proof.
    intros.

    apply X0. apply X. apply X1.
Qed.

Theorem and1imply : forall a b l, (valid (And a b) l)=
               true -> valid a l=true.
Proof.
    intros.

    unfold valid in H. fold valid in H.
```

```
    destruct (valid a l). reflexivity. inversion H.
Qed.

Theorem and2imply : forall a b l, (valid (And a b) l)=true ->
         valid b l=true.
Proof.
    intros. unfold valid in H. fold valid in H.

    destruct (valid a l). apply H. inversion H.
Qed.

Inductive pickElement : Pr -> Pr -> Pr -> Prop :=
    | Left : forall a b l r, pickElement a l r ->
       pickElement (And a b) (And l b) r
    | Right : forall a b l r, pickElement b l r ->
       pickElement (And a b) (And a l) r
    | Pick : forall a b, a=b ->
       pickElement a (Atom (Const 1)) a.

Theorem mergeStep : forall X XX Y YY l e1 e2 Q,
    pickElement X XX e1 ->
    pickElement Y YY e2 ->
    e1=e2 ->
    (valid XX l=true \/ valid YY l=true -> valid Q l=true) ->
    (valid X l=true \/ valid Y l=true -> valid (And e2 Q) l=true).
Proof.
    admit.
Qed.

Theorem mergeFinish: forall l,
     valid (Atom (Const 1)) l=true.
Proof.
    admit.
Qed.

Theorem mergePredicates: forall e a b c d l,
    e = b ->
    valid (And a (And e c)) l=true \/
    valid (And b d) l=true ->
    valid b l=true.
Proof.
```

```
    intros e a b c d l H.
    eapply validTransitive.
    eapply mergeStep.
        eapply Right. eapply Left. eapply Pick. reflexivity.
        eapply Left. eapply Pick. reflexivity. apply H.
    intros. eapply mergeFinish.

    intros. eapply and1imply. apply H0.
Qed.
```

Appendix D

C code for DPLL

```
     #include <stdio.h>
     #include <stdlib.h>

     #define VAR_COUNT 10

     // Assignments array, 0=unassigned, 1=False, 2=True
ℓD1  char assignments[VAR_COUNT];

     // Clause data structure (and watch var info)
ℓD2  struct clause {
      struct clause *next;
      char positive_lit[VAR_COUNT];
      char negative_lit[VAR_COUNT];
      char watch_var[VAR_COUNT];
      struct clause *watch_next[VAR_COUNT];
      struct clause *watch_prev[VAR_COUNT];
     } *clauses;

     // Heads of watch variable linked lists
ℓD3  struct clause *watches[VAR_COUNT];

     // FIFO Queue of assignments to do
ℓD4  struct assignments_to_do {
      struct assignments_to_do *next, *prev;
      int var;
      char value;
     int unit_prop;
     } *assignments_to_do_head, *assignments_to_do_tail;

     // Stack of assignments (same info as
     // assignments array above)
ℓD5  struct assignment_stack {
      struct assignment_stack *next;
      int var;
      char value;
      char unit_prop;
     } *stack;

     // Main sat solver algorithm
ℓD6  int solve()
     {
ℓD7   int backtrack = 0;
```

```
        // main loop--either pick a new variable
        // main loop--or flipping a variable
ℓD8     while (1) {
         int i, var, value, have_var, prop;
         struct assignments_to_do *todo;
         struct clause *clause;
         have_var = 0;
         if (backtrack) {
          // flipping var, remove assignment
          backtrack = 0;
          var = stack->var;
          value = stack->value;
          struct stack *n = stack->next;
          free(stack);
          stack = n;
          have_var = 1;
          assignments[var] = 0;
         } else {
          // Picking var--find unassigned var
          value = 1;
          for (i = 0; i < VAR_COUNT; ++i) {
           if (assignments[i]==0) {
            var = i;
            have_var = 1;
           }
          }
         }.

ℓD9      // If no var picked, we are done--found solution
         if (!have_var) return 1;
         todo = malloc(sizeof(struct assignments_to_do));
         todo->next = assignments_to_do_head;
         todo->prev = NULL;
         if (assignments_to_do_tail==NULL) {
          assignments_to_do_tail = todo;
         } else {
          assignment_to_do_head->prev = todo;
         }
         assignments_to_do_head = todo;
         todo->var = var;
         todo->value = value;
         todo->unit_prop = 0;
         printf("***** CHOOSING %d to be %d \n", var, value);

         // Process all variables in FIFO queue
         // (Processing may add more variables)
ℓD10     while (assignments_to_do_tail != NULL) {
          printf(" Processing var %d with value %d \n",
          assignments_to_do_tail->var,
          assignments_to_do_tail->value);
```

```
            // Pick var and remove from queue
            var = assignments_to_do_tail->var;
            value = assignments_to_do_tail->value;
            prop = assignments_to_do_tail->unit_prop;
            if (assignments_to_do_tail->prev==NULL) {
             free(assignments_to_do_tail);
             assignments_to_do_head = NULL;
             assignments_to_do_tail = NULL;
            } else {
             struct assignments_to_do *x = assignments_to_do_tail->prev;
             free(assignments_to_do_tail);
             assignments_to_do_tail = x;
             x->next = NULL;
            }

            // Make sure variable is unassigned
ℓD11        if (assignments[var]==0) {
             assignments[var] = value;
             printf(" Setting value \n");

             // Add assignment
             struct assignment_stack *s =
              malloc(sizeof(struct assignment_stack));
             s->next = stack;
             stack = s;
             s->var = var;
             s->value = value;
             s->unit_prop = prop;
             struct clause *nc;

             // Walk through all clauses watching this var
ℓD12         for (clause = watches[var]; clause; clause = nc) {
              printf(" Processing clause %x \n", clause);
              nc = clause->watch_next[var];
ℓD13          if ((value == 2 && clause->negative_lit[var]) ||
               (value == 1 && clause->positive_lit[var])) {

               // Case where assignment in this clause is not
               // satisfying--move watch to new variable
               printf(" Processing watch \n");
               int j;
               int non_watch;
               int has_non_watch = 0;
               int skip = 0;

               // find unassigned variable in clause
               // (or if clause has satisfying assignment)
```

```
ℓD14        for (j = 0; j < VAR_COUNT; ++j) {
             if (clause->positive_lit[j]) {
              if (assignments[j] == 2) skip = 1;
             }
             if (clause->negative_lit[j]) {
              if (assignments[j] == 1) skip = 1;
             }
             if (!assignments[j] && clause->watch_var[j]==0) {
              non_watch = j;
              has_non_watch = 1;
             }
            }
            if (skip) {
             // have satisfying assignment--do nothing
             printf(" Case 0:  Skipping \n");
            } else {
             if (has_non_watch) {
              // Found unassigned--move watch there
              printf(" Case 1:  moving watch
               %d %d \n", var, non_watch);
ℓD15          clause->watch_next[non_watch] =
               watches[non_watch];
              clause->watch_var[non_watch] = 1;
              if (watches[non_watch]) {
               watches[non_watch]->watch_prev[non_watch] =
                clause;
              }
              watches[non_watch] = clause;
              clause->watch_var[var] = 0;
              if (clause->watch_prev[var]) {
               clause->watch_prev[var]->watch_next[var] =
                clause->watch_next[var];
              } else {
               watches[var] = clause->watch_next[var];
              }
              if (clause->watch_next[var]) {
               clause->watch_next[var]->watch_prev[var] =
                clause->watch_prev[var];
              }
             } else {
              // No suitable unassigned--mark unit propagation
              printf(" Case 2:
               Adding unit prop \n");
ℓD16          int k, v, val;
```

```
                for (k = 0; k < VAR_COUNT; ++k) {
                 if (clause->watch_var[k] &&
                  assignments[k]==0) {
                  v = k;
                  if (clause->positive_lit[v]) {
                   val = 2;
                  }
                  if (clause->negative_lit[v]) {
                   val = 1;
                  }
                 }
                }
                struct assignments_to_do *todo =
                 malloc(sizeof(struct assignments_to_do));
                todo->next = assignments_to_do_head;
                todo->prev = NULL;
                if (assignments_to_do_tail==NULL) {
                 assignments_to_do_tail = todo;
                } else {
                 assignments_to_do_head->prev = todo;
                }
                assignments_to_do_head = todo;
                printf(" Queuing propagation
                 var %d to %d \n", v, val);
                todo->var = v;
                todo->value = val;
                todo->unit_prop = 1;
               }
              }
             }
            }
           } else if (assignments[var] != value) {

ℓD17        // Conflict between queued assignment and
           // current value--backtrack
           printf("*** Backtracking due to
            %d with %d *** \n", var, value);
           // Clean out all the todos
           while (assignments_to_do_head) {
            todo = assignments_to_do_head->next;
            free(assignments_to_do_head);
            assignments_to_do_head = todo;
           }
           assignments_to_do_tail = NULL;
```

```
          // Clean out assignments until we find a positive
          // that can be flipped to a negative
          while (stack && (stack->unit_prop ||
            stack->value==2)) {
           printf(" dropping %d with value
            %d prop %d \n", stack->var,
            stack->value, stack->unit_prop);
           struct assignment_stack *n = stack->next;
           assignments[stack->var] = 0;
           free(stack);
           stack = n;
          }
          if (stack==NULL) {
           // Nothing to flip--formula is unsat
           printf("Back track exit \n");
           return 0;
          }
          stack->value = 2;
          assignments[stack->var] = 2;
          backtrack = 1;
         }
        }
       }
      }

main()
{
    char line[5000];

    while (!feof(stdin)) {
        gets(line);
        if (isdigit(line[0]) || line[0]=='-') {
            printf("Adding clause %s\n", line);
            int i;
            struct clause *cl = malloc(sizeof(struct clause));
            printf("Allocating clause %x\n", cl);
            cl->next = clauses;
            clauses = cl;
            for (i = 0; i < VAR_COUNT; ++i) {
                cl->watch_next[i] = NULL;
                cl->watch_prev[i] = NULL;
                cl->watch_var[i] = 0;
                cl->positive_lit[i] = 0;
                cl->negative_lit[i] = 0;
            }
            char *c = line;
```

```
            int count = 0;
            while (*c=='-' || isdigit(*c)) {
                int x = atoi(c);
                while (*c=='-' || isdigit(*c)) ++c;
                while (*c==' ') ++c;
                if (x < 0) {
                    x = 0-x;
                    --x;
                    cl->negative_lit[x] = 1;
                } else {
                    --x;
                    cl->positive_lit[x] = 1;
                }
                if (count < 2) {
                    cl->watch_next[x] = watches[x];
                    if (watches[x]) {
                        watches[x]->watch_prev[x] = cl;
                    }
                    watches[x] = cl;
                    cl->watch_var[x] = 1;
                    ++count;
                }
            }
        }
    }
    int i;
    for (i = 0; i < VAR_COUNT; ++i) {
        printf("Watches for %d\n", i);
        struct clause *cl = watches[i];
        while (cl) {
            printf("    %x\n", cl);
            cl = cl->watch_next[i];
        }
    }
    if (solve()) {
        printf("SAT\n");
        int i;
        for (i = 0; i < VAR_COUNT; ++i) {
            printf("var %d is %d\n", i, assignments[i]);
        }
    } else {
        printf("UNSAT\n");
```

```
    }
    printf("Done\n");
}
```

Appendix E

PEDANTIC DPLL

```
      Definition Program :=
       backtrack ::= ANum(0);
ℓE1    WHILE ANum(1) DO
        have_var ::= ANum(0);
        IF (!backtrack) THEN
         backtrack ::= A0;
ℓE2      (CLoad varx (!stack+++ANum(stack_var_offset)));
         valuex ::= A2;
ℓE3      (CLoad ssss (!stack+++ANum(next_offset)));
ℓE4      DELETE !stack,ANum(sizeof_assignment_stack);
         stack ::= !ssss;
         have_var ::= A1;
ℓE5      (CStore (!assignments+++!varx) A0)
        ELSE
         valuex ::= A1;
         iiii ::= A0;
ℓE6      WHILE (!iiii <<= ANum(var_count)) DO
ℓE7       (CLoad ssss (!assignments+++!iiii));
ℓE8       IF (!ssss===A0) THEN
           varx ::= !iiii;
           have_var ::= A1
          ELSE
           SKIP
ℓE9       FI;
          iiii ::= !iiii +++ A1
ℓE10     LOOP
ℓE11    FI;
        IF (!have_var===A0) THEN
         RETURN A1
        ELSE
         SKIP
        FI;
        NEW todo,ANum(sizeof_assignments_to_do);
        (CStore (!todo+++ANum(next_offset))
          (!assignments_to_do_head));
        (CStore (!todo+++ANum(prev_offset)) A0);
        IF (!assignments_to_do_tail===A0) THEN
         assignments_to_do_tail ::= !todo
        ELSE
         (CStore (!assignments_to_do_head+++
           ANum(prev_offset)) (!todo))
        FI;
        assignments_to_do_head ::= !todo;
```

```
(CStore (!todo+++ANum(todo_var_offset)) (!varx));
(CStore (!todo+++ANum(todo_val_offset)) (!valuex));
(CStore (!todo+++ANum(todo_unit_offset)) A0);
WHILE !assignments_to_do_tail DO
 (CLoad varx (!assignments_to_do_tail+++
   ANum(todo_var_offset)));
 (CLoad valuex (!assignments_to_do_tail+++
   ANum(todo_val_offset)));
 (CLoad prop (!assignments_to_do_tail+++
   ANum(todo_unit_offset)));
 (CLoad ssss (!assignments_to_do_tail+++
   ANum(prev_offset)));
 IF !ssss THEN
  assignments_to_do_tail ::= !ssss;
  (CStore (!ssss+++ANum(next_offset-1)) A0);
  DELETE !ssss,(ANum(sizeof_assignments_to_do))
 ELSE
  DELETE !ssss,(ANum(sizeof_assignments_to_do));
  assignments_to_do_head ::= A0;
  assignments_to_do_tail ::= A0
 FI;
 (CLoad ssss (!assignments+++!varx));
 IF !ssss THEN
  (CIf ((!ssss)===(!valuex))
   (SKIP)
  (*ELSE*)
   (WHILE (!assignments_to_do_head) DO
    (CLoad todo (!assignments_to_do_head +++
      ANum(next_offset)));
    DELETE !assignments_to_do_head,
      ANum(sizeof_assignments_to_do);
    assignments_to_do_head ::= !todo
  LOOP;
  assignments_to_do_tail ::= A0;
  (CLoad ssss (!stack +++ ANum(stack_prop_offset)));
  (CLoad vvvv (!stack +++ ANum(stack_val_offset)));
  WHILE (ALand (!stack) (ALor (!ssss) (!vvvv===A2))) DO
   (CLoad kkkk (!stack +++ ANum(next_offset)));
   (CLoad vvvv (!stack +++ ANum(stack_var_offset)));
   (CStore (!assignments +++ !vvvv) A0);
   DELETE !stack,ANum(sizeof_assignment_stack);
   stack ::= !kkkk;
   (CLoad ssss (!stack +++ ANum(stack_prop_offset)));
   (CLoad vvvv (!stack +++ ANum(stack_val_offset)))
  LOOP;
```

```
 IF (!stack===A0) THEN
  RETURN A0
 ELSE
  SKIP
 FI;
 (CStore (!stack +++ ANum(stack_val_offset-1)) A2);
 (CLoad vvvv (!stack +++ ANum(stack_var_offset)));
 backtrack ::= A1))
ELSE
 (CStore (!assignments+++!varx) (!valuex));
 NEW ssss,ANum(sizeof_assignment_stack);
 (CStore (!ssss+++ANum(next_offset-1)) (!stack));
 stack ::= !ssss;
 (CStore (!ssss+++ANum(stack_var_offset-1)) (!varx));
 (CStore (!ssss+++ANum(stack_val_offset-1)) (!valuex));
 (CStore (!ssss+++ANum(stack_prop_offset-1)) (!prop));
 (CLoad clause (!watches+++!varx));
 WHILE (!clause) DO
  (CLoad nc (!clause+++ANum(watch_next_offset)+++
    !varx));
  (CLoad ssss (!clause+++ANum(negative_lit_offset)+++
    !varx));
  (CLoad vvvv (!clause+++ANum(positive_lit_offset)+++
    !varx));
  IF (ALor (ALand (!valuex === A2) (!ssss))
    (ALand (!valuex === A1) (!vvvv))) THEN
   has_non_watch ::= A0;
   skip ::= A0;
   jjjj ::= A0;
   WHILE (!jjjj <<= ANum(var_count-1)) DO
    jjjj ::= !jjjj +++ A1;
    (CLoad ssss (!clause+++
      ANum(positive_lit_offset)+++!jjjj));
    (CLoad vvvv (!assignments+++!jjjj));
    IF (ALand (!ssss) (!vvvv===A2)) THEN
     skip ::= A1
    ELSE
     SKIP
    FI;
    (CLoad ssss (!clause+++
      ANum(negative_lit_offset)+++!jjjj));
    IF (ALand (!ssss) (!vvvv===A1)) THEN
     skip ::= A1
    ELSE
     SKIP
    FI;
    (CLoad ssss (!clause+++
      ANum(watch_var_offset)+++!jjjj));
    IF (ALand (!vvvv===A0) (!ssss===A0)) THEN
```

```
  (CLoad ssss (!clause+++
    ANum(positive_lit_offset)+++!jjjj));
  (CLoad vvvv (!clause+++
    ANum(negative_lit_offset)+++!jjjj));
  IF (ALor (!ssss) (!vvvv)) THEN
   non_watch ::= !jjjj;
   has_non_watch ::= A1
  ELSE
   SKIP
  FI
 ELSE
  SKIP
 FI
LOOP;
IF (!skip) THEN
 SKIP
ELSE
 SKIP;
 IF (!has_non_watch) THEN
  (CLoad ssss (!watches +++ !non_watch));
  (CStore (!clause +++ ANum(watch_next_offset)
    +++ !non_watch) (!ssss));
  (CStore (!clause +++ ANum(watch_var_offset)
    +++ !non_watch) A1);
  IF (!ssss) THEN
   (CStore (!ssss +++
    ANum(watch_prev_offset) +++ !non_watch) (!clause))
  ELSE
   SKIP
  FI;
  (CStore (!watches +++ !non_watch) (!clause));
  (CStore (!clause +++
    ANum(watch_var_offset) +++ !varx) A0);
  (CLoad ssss (!clause +++
    ANum(watch_prev_offset) +++ !varx));
  (CLoad vvvv (!clause +++
    ANum(watch_next_offset) +++ !varx));
  IF (!ssss) THEN
   (CStore (!ssss +++ ANum(watch_next_offset)
     +++ !varx) (!vvvv))
  ELSE
   (CStore (!watches +++ !varx) (!vvvv))
  FI;
 I F (!vvvv) THEN
   (CStore (!vvvv +++
    ANum(watch_prev_offset) +++ !varx) (!ssss))
  ELSE
   SKIP
  FI
 ELSE
  kkkk ::= A0;
```

```
            WHILE (!kkkk <<= ANum(var_count)) DO
             (CLoad ssss (!clause
               +++ ANum(watch_var_offset) +++ !kkkk));
             (CLoad jjjj (!assignments +++ !kkkk));
             IF (ALand (!ssss) ((!jjjj)===A0)) THEN
              vvvv ::= !kkkk;
              (CLoad ssss (!clause
                +++ ANum(positive_lit_offset) +++ !vvvv));
              IF (!ssss) THEN
               val ::= A2
              ELSE
               SKIP
              FI;
              (CLoad ssss (!clause +++
                ANum(negative_lit_offset) +++ !vvvv));
              IF (!ssss) THEN
               val ::= A1
              ELSE
               SKIP
              FI
             ELSE
              SKIP
             FI;
             kkkk ::= !kkkk +++ A1
            LOOP;
            NEW todo,ANum(sizeof_assignments_to_do);
            (CStore (!todo +++ ANum(next_offset))
              (!assignments_to_do_head));
            (CStore (!todo +++ ANum(prev_offset)) A0);
            IF (!assignments_to_do_tail) THEN
             (CStore (!assignments_to_do_head +++
               ANum(prev_offset)) (!todo))
            ELSE
             assignments_to_do_tail ::= !todo
            FI;
            assignments_to_do_head ::= !todo;
            (CStore (!todo +++ ANum(todo_var_offset-1)) (!vvvv));
            (CStore (!todo +++ ANum(todo_val_offset-1)) (!val));
            (CStore (!todo +++ ANum(todo_unit_offset-1)) A1)
           FI
          FI
         ELSE
          SKIP
         FI;
         clause ::= !nc
        LOOP
       FI
      LOOP
     LOOP.
```

Appendix F

DPLL Invariant

```
Notation "'clauses'" := (Id 1001) (at level 1).
Notation "'assignments_to_do_head'" := (Id 1002) (at level 1).
Notation "'assignments_to_do_tail'" := (Id 1003) (at level 1).
Notation "'stack'" := (Id 1004) (at level 1).
Notation "'assignments'" := (Id 1005) (at level 1).
Notation "'watches'" := (Id 1006) (at level 1).
Notation "'backtrack'" := (Id 1007) (at level 1).
Notation "'iiii'" := (Id 1008) (at level 1).
Notation "'varx'" := (Id 1009) (at level 1).
Notation "'valuex'" := (Id 1010) (at level 1).
Notation "'have_var'" := (Id 1011) (at level 1).
Notation "'prop'" := (Id 1012) (at level 1).
Notation "'todo'" := (Id 1013) (at level 1).
Notation "'clause'" := (Id 1014) (at level 1).
Notation "'ssss'" := (Id 1015) (at level 1).
Notation "'vvvv'" := (Id 1016) (at level 1).
Notation "'val'" := (Id 1017) (at level 1).
Notation "'kkkk'" := (Id 1018) (at level 1).
Notation "'nc'" := (Id 1019) (at level 1).
Notation "'jjjj'" := (Id 1020) (at level 1).
Notation "'non_watch'" := (Id 1021) (at level 1).
Notation "'has_non_watch'" := (Id 1022) (at level 1).
Notation "'skip'" := (Id 1022) (at level 1).

Definition var_count := 4.

Definition next_offset := 0.
Definition positive_lit_offset := 1.
Definition negative_lit_offset := var_count + 1.
Definition watch_var_offset := var_count * 2 + 1.
Definition watch_next_offset := var_count * 3 + 1.
Definition watch_prev_offset := var_count * 4 + 1.

Definition sizeof_clause := var_count * 5 + 1.

Definition prev_offset := 1.
Definition todo_var_offset := 2.
Definition todo_val_offset := 3.
Definition todo_unit_offset := 4.

Definition sizeof_assignments_to_do := 5.

Definition stack_var_offset := 1.
Definition stack_val_offset := 2.
Definition stack_prop_offset := 3.
```

```
Definition sizeof_assignment_stack := 4.

Definition level := 0.
Definition domain (x :  absExp) :  absExp := #0.
```

ℓ_{F1}

```
Definition coreStructures :  absState
  (*clauses assignments_to_do_head,stack,assignments,watches*) :=
 TREE(v(5),v(0),#sizeof_clause,(#(next_offset)::nil)) **
 TREE(v(6),v(1),
  #sizeof_assignments_to_do,(#(next_offset)::nil))**
 TREE(v(7),v(2),#sizeof_assignment_stack,(#(next_offset)::nil))**
 ARRAY(v(8),#var_count,v(3)) **
 ARRAY(v(9),#var_count,v(4)).
```

ℓ_{F2}

```
Definition treeInArray (* tr,ar,l *) :  absState :=
 (AbsAll TreeRecords(v(0))
  ([--(v(S(0)),v(0))-->stack_var_offset <<<< #var_count] **
   ([--(v(S(0)),v(0))-->stack_val_offset ==== #1] *\/*
    [--(v(S(0)),v(0))-->stack_val_offset ==== #2]) **
    ([nth(v(S(1)),--(v(S(0)),v(0))-->stack_var_offset)====
     --(v(S(0)),v(0))-->stack_val_offset]) **
     (AbsAll TreeRecords(nth(find(v(S(0)),v(0)),#1))
      ([  (--(v(S(S(0))),v(1))-->stack_var_offset====
       --(nth(find(v(S(S(0))),v(1)),#1),v(0))
        -->stack_var_offset)]))))).
```

ℓ_{F3}

```
Definition arrayInTree (* tr ar *) :  absState :=
 (AbsAll range(#0,#(var_count))
  ([nth(v(S(1)),v(0))====#0] *\/*
  AbsExists (TreeRecords(v(S(0))))
   ([(--(v(S(S(0))),v(0))-->stack_var_offset====v(1) //\\
    --(v(S(S(0))),v(0))-->stack_val_offset====
     nth(v(S(S(1))),v(1)))]) )).
```

ℓ_{F4}

```
Definition treeEquivArray (* tr ar l *) :  absState :=
 treeInArray **
 arrayInTree.

Definition validBackPointers (* tr *)
  prev_offset next_offset :  absState :=
 (AbsAll TreeRecords(v(0))
  ([
   (--(v(S(0)),v(0))--->(prev_offset)====#0 //\\
    (nth(v(S(0)),v(0))====v(0)) \\//
   (--(v(S(0)),v(0))--->(prev_offset) inTree v(0) //\\
   --(v(S(0)),--(v(S(0)),v(0))--->(prev_offset))
    --->(next_offset)====v(0)))
  ])).
```

```
Definition satisfyingAssignmentMade (*tr c a*) :  absState :=
 (AbsExists range(#0,#(var_count))
  ([(--(v(1),v(2))--->(#positive_lit_offset++++v(0)) //\\
   nth(v(3),v(0))====#2) \\//
   (--(v(1),v(2))--->(#negative_lit_offset++++v(0)) //\\
    nth(v(3),v(0))====#1)])).

Definition satisfyingAssignmentMadeExcept
   (*tr c a v*) :  absState :=
 (AbsExists range(#0,#(var_count))
  (([   (v(4)====v(0))]) **
   ([(--(v(1),v(2))--->(#positive_lit_offset++++v(0)) //\\
    nth(v(3),v(0))====#2) \\//
   (--(v(1),v(2))--->(#negative_lit_offset++++v(0)) //\\
    nth(v(3),v(0))====#1)]))).

Definition atleastTwoUnassigned (*cl a b v*) :  absState :=
 (AbsAll TreeRecords(v(0))
  ((((AbsClosure satisfyingAssignmentMadeExcept
   (v(1)::v(0)::v(2)::v(4)::v(3)::nil)) *\/* ([(v(3)====#0)])) **
 ((AbsClosure satisfyingAssignmentMade
  (v(1)::v(0)::v(2)::nil)))) *\/*
  (AbsExists range(#0,#(var_count))
   (AbsExists range(#0,#(var_count))
    (([  (v(0)====v(1))]) **
   ([((--(v(3),v(2))--->(#positive_lit_offset++++v(0))) \\//
    (--(v(3),v(2))--->(#negative_lit_offset++++v(0)))) //\\
    nth(v(4),v(0))====#0]) **
    ([((--(v(3),v(2))--->(#positive_lit_offset++++v(1))) \\//
     (--(v(3),v(2))--->(#negative_lit_offset++++v(1)))) //\\
     nth(v(4),v(1))====#0])))))).
```

```
Definition atleastTwoUnassignedOrLast (*cl a st*) :  absState :=
 (AbsAll TreeRecords(v(0))
  (
   (AbsExists range(#0,#(var_count))
    ([((--(v(2),v(1))--->(#positive_lit_offset++++v(0))) \\//
     (--(v(2),v(1))--->(#negative_lit_offset++++v(0)))) //\\
     nth(v(3),v(0))====#0]) **
    ([(--(v(2),v(1))--->(#positive_lit_offset++++
     nth(v(4),#stack_var_offset))) \\//
   (--(v(2),v(1))--->(#negative_lit_offset++++
    nth(v(4),#stack_var_offset)))])
   ) *\/*
   (AbsExists range(#0,#(var_count))
    (AbsExists range(#0,#(var_count))
     (([  (v(0)====v(1))]) **
    ([((--(v(3),v(2))--->(#positive_lit_offset++++v(0))) \\//
    (--(v(3),v(2))--->(#negative_lit_offset++++v(0)))) //\\
    nth(v(4),v(0))====#0]) **
     ([((--(v(3),v(2))--->(#positive_lit_offset++++v(1))) \\//
      (--(v(3),v(2))--->(#negative_lit_offset++++v(1)))) //\\
     nth(v(4),v(1))====#0])))))).

Definition twoUnassignedAtChoiceSteps (*cl a st*) :  absState :=
 (AbsAll TreeRecords(v(2))
  (AbsAll TreeRecords(v(0))
   (([  (#0 ==== (--(v(3),v(0))-->stack_prop_offset))]) *\/*
    (AbsAll TreeRecords(v(2))
     (([find(v(4),v(3))====v(4)]) *\/*
     (AbsExists range(#0,#(var_count))
      (AbsExists range(#0,#(var_count))
       (([nth(v(7),v(0))====#0 //\\
       nth(v(7),v(1))==== #0 //\\   (v(0)====v(1))]) **
       ([  (--(v(5),v(2))--->(#positive_lit_offset ++++
        v(1))====#0) \\//
          (--(v(5),v(2))--->(#negative_lit_offset ++++
        v(1))====#0)]) **
       ([  (--(v(5),v(2))--->(#positive_lit_offset ++++
        v(0))====#0) \\//
          (--(v(5),v(2))--->(#negative_lit_offset ++++
        v(0))====#0)]))))
 ))))).
```

```
Definition assignmentConsistent (*cl c a*) :  absState :=
 (AbsExists range(#0,#(var_count))
  ([
   (--(v(1),v(2))--->(#positive_lit_offset++++v(0)) //\\
   (nth(v(3),v(0))====#2 \\// nth(v(3),v(0))====#0)) \\//
    (--(v(1),v(2))--->(#negative_lit_offset++++v(0)) //\\
     (nth(v(3),v(0))====#1 \\// nth(v(3),v(0))====#0))
 ])).

Definition watchVariablesExists (*tr c*) :  absState :=
 (AbsAll range(#0,#(var_count))
  ([
   (--(v(1),v(2))--->(#watch_var_offset++++v(0))====#0) \\//
    (--(v(1),v(2))--->(#positive_lit_offset++++v(0))) \\//
    (--(v(1),v(2))--->(#negative_lit_offset++++v(0)))])).

Definition watchVariablesLinkedIffSet (*tr c w*) :  absState :=
 (AbsAll range(#0,#(var_count))
  ([
   (  (--(v(1),v(2))--->(#watch_var_offset++++v(0))====#0) //\\
    (  (--(v(1),v(2))--->(#watch_prev_offset++++v(0))====#0) \\//
     nth(v(3),v(0))====v(2))) \\//
    (--(v(1),v(2))--->(#watch_var_offset++++v(0))====#0 //\\
     --(v(1),v(2))--->(#watch_prev_offset++++v(0))====#0 //\\
        (nth(v(3),v(0))====(v(2))))])).

Definition twoWatchVariables (*tr c*) :  absState :=
 (SUM(range(#0,#(var_count)),ite((--(v(1),v(2))--->
  (#watch_var_offset++++v(0))),(#1),(#0)),#2)).

Definition onlyOneUnassigned (*tr c a*) :  absState :=
 SUM(range(#0,#(var_count)),
  (((--(v(1),v(2))--->(#positive_lit_offset++++v(0)))
   \\// (--(v(1),v(2))--->(#negative_lit_offset++++v(0)))) //\\
   ite(nth(v(3),v(0))====#0,#1,#0)),
   #1).
```

```
Definition unassignedVariablesAreWatches (*tr c a*):absState :=
 (AbsAll range(#0,#(var_count))
  ([(#0<<<<--(v(1),v(2))--->(#watch_var_offset++++v(0)) //\\
   (nth(v(3),v(0))====#0)) \\//
   (
    ((#0<<<<nth(v(3),v(0))) \\// ((--(v(1),v(2))--->
     (#positive_lit_offset++++v(0))====#0 //\\
    --(v(1),v(2))--->(#negative_lit_offset++++v(0))====#0))))])).

Definition mostRecentAssignedIsWatch (*tr c a st*) :absState :=
 (AbsAll range(#0,#(var_count))
  (AbsAll range(#0,#(var_count))
   (([--(v(2),v(3))--->(#watch_var_offset++++v(0)) \\//
    ((((--(v(2),v(3))--->(#positive_lit_offset++++v(0)))====#0
     //\\ (--(v(2),v(3))--->(#negative_lit_offset++++v(0))
     ====#0)))) \\//
       (--(v(2),v(3))--->(#watch_var_offset++++v(1))) \\//
     nth(v(4),v(0))====#0 \\// nth(v(4),v(1))====#0
      \\// v(0)====v(1)])
    (AbsExists TreeRecords(v(5))
     (([--(v(6),v(0))-->stack_var_offset====v(1)]) **
     (AbsExists TreeRecords(find(v(6),v(0)))
      ([--(v(7),v(0))-->stack_var_offset====v(3)])))))))).

Definition allButOneAssigned (*tr c a st*) :  absState :=
 onlyOneUnassigned **
 unassignedVariablesAreWatches **
 mostRecentAssignedIsWatch.

Definition watchAfterSatisfyingAssignment (*tr c a st*) :
  absState :=
 (AbsAll range(#0,#(var_count))
  ([#0====nth(v(3),v(0))] *\/*
  ([--(v(1),v(2))--->(#watch_var_offset++++v(0))====#0] **
   [#0<<<<nth(v(3),v(0))]) *\/*
  (AbsExists TreeRecords(v(4))
   ([--(v(5),v(0))-->stack_var_offset====v(1)]) **
   (AbsExists TreeRecords(find(v(5),v(0)))
    ([
    ((#0 <<<< (--(v(3),v(4))--->
     (#positive_lit_offset++++ --(v(6),v(0))-->stack_var_offset)))
     //\\ (--(v(6),v(0))-->stack_val_offset====#2))
    \\//
    ((#0 <<<< (--(v(3),v(4))--->(#negative_lit_offset++++
     --(v(6),v(0))-->stack_var_offset))) //\\
     (--(v(6),v(0))-->stack_val_offset====#1))
   ]))
 ))).
```

```
      Definition watchesUnassigned (*tr c a*) :  absState :=
       (AbsAll range(#0,#(var_count))
        ([--(v(1),v(2))--->(#watch_var_offset++++v(0))====#0 \\//
         nth(v(3),v(0))====#0])).

      Definition validTail (*t v*) :  absState :=
       ([v(2)====#0] ** [v(1)====#0]) *\/*
         ([v(2) inTree v(0)] **
         [--(v(2),v(0))-->next_offset====#0]).

      Definition invariantCoreNoTail
         (*v0 v1 v2 v3 v4 clauses assignments_to_do stack assignments
         watches backtrack assignments_to_do_tail have_var varx *):
         absState := (
ℓF5     (AbsClosure coreStructures (v(0)::v(1)::v(2)::v(3)::v(4)::
         v(5)::v(6)::v(7)::v(8)::v(9)::nil)) **
        (* Assertions that the stack and assignments array contain
         the same set of assignments *)
ℓF6     (AbsClosure treeEquivArray (v(2)::v(3)::nil)) **
        (* Assertion defining the prev pointer in the
         assignments_to_do
         doubly linked list--actually, this list should be empty *)
ℓF7     ([v(6)==== #0 //\\ v(11)==== #0]) **
ℓF8     (* (AbsClosure (validBackPointers (#prev_offset)
         (#next_offset)) (v(1)::nil)) ** *)
ℓF9     ((([  (v(10) ==== #0)]) ** (AbsClosure
         atleastTwoUnassignedOrLast (v(0)::v(3)::v(2)::nil)))
        *\/* (AbsClosure atleastTwoUnassigned
         (v(0)::v(3)::v(10)::v(13)::nil))) **
ℓF10    (AbsClosure twoUnassignedAtChoiceSteps
         (v(0)::v(3)::v(2)::nil)) **
ℓF11    (AbsEach range(#0,#(var_count))
         (* Define the basic linked list connecting the watch
          variables inside the clauses linked list *)
         (AbsExistsT
ℓF12      ((Path((nth(v(6),v(1))), v(2), v(0), #sizeof_clause,
         ((#watch_next_offset++++v(1))::nil))) **
         (* Define the prev variable and the fact that if null we
          are atthe head of the list *)
ℓF13     (AbsClosure (validBackPointers (#watch_prev_offset++++v(1))
          (#watch_next_offset++++v(1))) (v(0)::nil)))) **
         (AbsAll TreeRecords(v(0))
          (* The current assignment is consistent with the clause *)
ℓF14      ((AbsClosure assignmentConsistent
           (v(1)::v(0)::v(4)::nil)) **
```

```
          (*
           * make sure that if the watch_var field is non-zero
           * (pointing to a variable) that watch_next and watch_prev
           * put this clause into the linked list for the watch
           * variable.  Also, for all watch variables, either
           * positive_lit or negative_lit is true.
           *)
ℓF15      (AbsClosure watchVariablesExists (v(1)::v(0)::nil)) **
ℓF16      (AbsClosure watchVariablesLinkedIffSet
           (v(3)::v(0)::v(7)::nil)) **
          (* Make sure there are precisely two watch variables per
           clause or all variables are watches, needs fixing?  *)
ℓF17      (AbsClosure twoWatchVariables (v(1)::v(0)::nil)) **
          (* Watch variable invariant--case 1:  All but one variable
           in the clause are assigned, any watch variable pointing
           to an assigned variable is pointing to a variable that
           was assigned after all other assigned variables in the
           clause.  Also, one of the two watch variables points to
           the one unassigned variable *)
ℓF18      ((AbsClosure allButOneAssigned
           (v(1)::v(0)::v(4)::v(3)::nil)) *\/*
          (* Watch variable invariant case 2:  One of the
           assignments already satisfies the clause, if a watch
           variable is assigned a value,then that value must be a
           satisfying assignment or occurred after a satisfying
           assignment *)
ℓF19      ( (AbsClosure satisfyingAssignmentMade
           (v(1)::v(0)::v(4)::nil)) **
ℓF20       (AbsClosure watchAfterSatisfyingAssignment
           (v(1)::v(0)::v(4)::v(3)::nil))) *\/*
          (* Watch variable invariant case 3:  both watch variables
           point to unassigned variables *)
ℓF22      (AbsClosure
           watchesUnassigned (v(1)::v(0)::v(4)::nil)))))).

        Definition invariantCore:  absState :=
         invariantCoreNoTail **
         (AbsClosure validTail (v(1)::v(6)::v(11)::nil)).

ℓF22    Definition loopInvariant (* clauses assignments_to_do stack
          assignments watches backtrack *) :  absState :=
         (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT (AbsExistsT
          (AbsClosure (invariantCore ** ([(#0) <<<< v(7)] *\/*
          [v(10) ==== (#0)])) (v(0)::v(1)::v(2)::v(3)::v(4)::
          (!!clauses)::(!!assignments_to_do_head)::(!!stack)::
          (!!assignments)::(!!watches)::(!!backtrack)::
          (!!assignments_to_do_tail)::(!!have_var)::
          (!!varx)::nil))))))).
```

Appendix G

Top level proof of DPLL algorithm

```
      Theorem SatProgramWorks :
       exists x, {{(AbsClosure invariant ((!!clauses)::
      (!!assignments_to_do_head)::(!!stack)::(!!assignments)::
      (!!watches)::nil))}}
      Program
      {{(finalState x) return (#0::#1::nil) with (finalState x)}}.
      Proof.
ℓG1    unfold Program.
       eapply ex_intro.
       eapply strengthenPost.
       eapply compose.
ℓG2    pcrunch.
ℓG3    eapply whileThm with (invariant := loopInvariant).
       instantiate ( 1:= invariant).

ℓG4    eapply strengthenPost.
       pcrunch.
ℓG5    eapply preCond1.  apply H0.

ℓG6    eapply preCond2.  apply (fun x => 0).  apply H. apply H0.

ℓG7    eapply whileThm with (invariant := haveVarInvariant).
       instantiate ( 1 := invariant ).

ℓG8    eapply strengthenPost.
       pcrunch.
ℓG9    apply mergeTheorem1.
ℓG10   apply entailment1.

       intros.  inversion H. intros.  inversion H.

ℓG11   apply entailment2.  apply (fun x => 0, fun x => None).
       instantiate (1 := invariant).
ℓG12   apply mergeTheorem2.
ℓG13   eapply validRefTheorem1.
        apply (Id 0).  apply n.  apply H. apply H0.
ℓG14   eapply validRefTheorem2.
        apply (Id 0).  apply n.  apply H. apply H0.
ℓG15   eapply validRefTheorem3.
        apply (Id 0).  apply n.  apply H. apply H0.
       instantiate (1 := invariant).
```

ℓ_{G16}

```
apply mergeTheorem3.
eapply validRefTheorem4.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem5.
 apply (Id 0).  apply n.  apply H. apply H0.

admit.

eapply while with (invariant :=invariant).
 eapply sbasic.
instantiate (1 := invariant).
 instantiate (1 := (#0::#1::nil)).
eapply strengthenPost.  pcrunch.

eapply validRefTheorem7.
 apply (Id 0).  apply n.  apply H. apply H0.

apply existsTheorem1.
apply existsTheorem2.

instantiate (1 := invariant).
apply mergeTheorem4.

eapply while with (invariant := invariantNoTail).
 eapply sbasic.
eapply strengthenPost.  pcrunch.
eapply existsTheorem3.
apply entailment3.

intros.  inversion H. intros.  inversion H.
apply entailment4.

pcrunch.

eapply while with (invariant := invariant).
 eapply sbasic.
eapply strengthenPost.  pcrunch.

eapply validRefTheorem8.
 apply (Id 0).  apply n.  apply H. apply H0.
apply existsTheorem4.
apply entailment5.

intros.  inversion H.
intros.  inversion H.
apply entailment6.
eapply validRefTheorem9.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem10.
 apply (Id 0).  apply n.  apply H. apply H0.
```

```
instantiate (1 := (#0::#1::nil)).
 instantiate (1 := (#0::#1::nil)).
instantiate (1 := invariant).
 instantiate (1 := invariant).
apply mergeReturn1.
instantiate (1 := (#0::#1::nil)).
 instantiate (1 := (#0::#1::nil)).
instantiate (1 := invariant).
 instantiate (1 := invariant).
apply mergeReturn2.

instantiate (1 := invariant).
apply mergeTheorem5.
eapply validRefTheorem11.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem12.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem13.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem14.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem15.
 apply (Id 0).  apply n.  apply H. apply H0.

eapply while with (invariant := invariantWL).
 eapply sbasic.
eapply strengthenPost.  pcrunch.

eapply while with (invariant := invariantWLIT1).
 eapply sbasic.
eapply strengthenPost.  pcrunch.

instantiate (1 := invariantWLIT1).
apply mergeTheorem6.

instantiate (1 := invariantWLIT1).
apply mergeTheorem7.

instantiate (1 := invariantWLIT1).
apply mergeTheorem8.

intros.  apply H.

intros.  inversion H.
intros.  inversion H.
```

```
apply entailment7.
eapply validRefTheorem16.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem17.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem18.
 apply (Id 0).  apply n.  apply H. apply H0.

instantiate (1 := invariantWLIT1).
apply mergeTheorem9.

eapply validRefTheorem19.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem20.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem21.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem22.
 apply (Id 0).  apply n.  apply H. apply H0.

instantiate (1 := invariantWLIT1).
apply mergeTheorem10.

eapply validRefTheorem23.
 apply (Id 0).  apply n.  apply H. apply H0.

instantiate (1 := invariantWLIT1).
apply mergeTheorem11.

eapply while with (invariant := invariantWLIT2).
 eapply sbasic.
eapply strengthenPost.  pcrunch.

instantiate (1 := invariantWLIT2).
apply mergeTheorem12.

instantiate (1 := invariantWLIT2).
apply mergeTheorem13.

instantiate (1 := invariantWLIT2).
apply mergeTheorem14.

apply entailment8.

intros.  inversion H.
intros.  inversion H.

apply entailment9.
```

```
eapply validRefTheorem24.  apply (Id 0).
 apply n.  apply H. apply H0.
eapply validRefTheorem25.  apply (Id 0).
 apply n.  apply H. apply H0.
eapply validRefTheorem26.  apply (Id 0).
 apply n.  apply H. apply H0.

instantiate (1 := invariantWLIT2).
apply mergeTheorem15.

eapply validRefTheorem27.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem28.
 apply (Id 0).  apply n.  apply H. apply H0.
eapply validRefTheorem29.
 apply (Id 0).  apply n.  apply H. apply H0.

instantiate (1 := invariantWLIT1).
apply mergeTheorem16.

instantiate (1 := invariantWLIT1).
apply mergeTheorem17.

instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := invariant).
instantiate (1 := invariant).
instantiate (1 := invariant).
apply mergeReturn3.

instantiate (1 := invariantWL).
apply mergeTheorem18.

apply entailment10.

intros.  apply H.
instantiate (1 := (#0::#1::nil)).
apply equivEvalList1.

apply entailment11.

instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := invariant).
apply mergeReturn4.
instantiate (1 := invariant).
apply mergeTheorem19.
```

```
intros.  apply H.
intros.  apply H.

apply equivEvalList1.

apply entailment12.

instantiate (1 := (#0)::(#1)::nil).
 instantiate (1 := invariant).
apply mergeReturn5.

instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := invariant).
apply mergeReturn6.

apply entailment13.

intros.  apply H.

apply equivEvalList1.

apply entailment14.

instantiate (1 := (#0)::(#1)::nil).
instantiate (1 := invariant).
apply mergeReturn7.  instantiate (1 := 0).

apply entailment15.
apply entailment16.
intros.  apply equivEvalList1.  apply H. apply H.

Grab Existential Variables.

apply (nil).  apply (nil).  apply (nil).  apply (nil).
apply (nil).  apply (nil).  apply (nil).  apply (nil).
apply (nil).  apply (nil).  apply (nil).  apply (nil).
apply (nil).  apply (nil).  apply (nil).

admit.
Admitted.
```

Bibliography

[1] Coq 8.5 beta release. `https://coq.inria.fr/news/123.html`. Accessed: 2015-03-20.

[2] Coqoon home page. `https://itu.dk/research/tomeso/coqoon/`. Accessed: 2015-03-19.

[3] MIT proofs page. `http://proofs.csail.mit.edu/`. Accessed: 2015-03-19.

[4] Peacoq home page. `http://goto.ucsd.edu/peacoq/`. Accessed: 2015-03-19.

[5] Proof general. `http://proofgeneral.inf.ed.ac.uk/`. Accessed: 2015-03-20.

[6] The heartbleed bug. `http://heartbleed.com/`, 2014.

[7] J. Alama, L. Mamane, and J. Urban. Dependencies in formal mathematics: Applications and extraction for Coq and Mizar. In *AISC/MKM/Calculemus*, pages 1–16, 2012.

[8] A. Appel and S. 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.

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

[10] A. W. Appel. Verified software toolchain. In *ESOP 2011: 20th European Symposium on Programming*, number LNCS 6602, pages 1–17, March 2011.

[11] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*, pages 135–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] N. Ayache. Combining the Coq proof assistant with first-order decision procedures, 2006.

[13] B. Barras, C. Tankink, and E. Tassi. Asynchronous processing of coq documents: From the kernel up to the user interface. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, pages 51–66, Cham, 2015. Springer International Publishing.

[14] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - a framework for higher-order separation logic in Coq. In *Third International Conference, ITP*, 2012.

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

[16] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of openssl hmac. In *USENIX Security*, volume 15, pages 207–221, 2015.

[17] Y. Bertot. Pcoq: A graphical user-interface for coq. `https://www-sop.inria.fr/lemme/pcoq/`.

[18] Y. Bertot. The ctcoq system: Design and architecture. *Formal Asp. Comput.*, 11(3):225–243, 1999.

[19] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[20] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[21] O. Boite. Proof reuse with extended inductive types. In *TPHOLs*, pages 50–65, 2004.

[22] T. Braibant and D. Pous. *Tactics for Reasoning Modulo AC in Coq*, pages 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[23] E. T. Bruno Barras, Carst Tankink. Asynchronous processing of coq documents: from the kernel up to the user interface. In *Proceedings of ITP*, 2015.

[24] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association of Computing Machinery*, 24(1):44–67, 1977.

[25] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible os kernels and device drivers. In *PLDI*, pages 431–447, 2016.

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

[27] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. 2009.

[28] D. Costanzo, Z. Shao, and Ronghui. End-to-end verification of information-flow security for c and assembly programs. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, June 2016.

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

[30] J. Darlington and R. Burstall. A system which automatically improves programs. *Acta Inf.*, 6:41–60, 1976.

[31] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. *The Lean Theorem Prover (System Description)*, pages 378–388. Springer International Publishing, Cham, 2015.

[32] N. Dershowitz. Orderings for term-rewriting systems. In *Theor. Comput. Sci. 17*, 1982.

[33] N. Dershowitz. Termination. In *RTA*, pages 180–224, 1985.

[34] N. Dershowitz. Termination of rewriting. In *J. Symb. Comput. 3(1/2)*, pages 69–116, 1987.

[35] N. Dershowitz and S. Kaplan. Rewrite, rewrite, rewrite, rewrite, rewrite. In *POPL*, pages 250–259, 1989.

[36] N. Dershowitz and S. Mitra. *Path orderings for termination of associative-commutative rewriting*, pages 168–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[37] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *FGCS*, pages 337–346, 1988.

[38] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *CTRS*, pages 31–44, 1987.

[39] J. V. Deshmukh, E. A. Emerson, and P. Gupta. Automatic verification of parameterized data structures. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 27–41, 2006.

[40] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[41] E. B. et al. Smtcoq: A plug-in for integrating smt solvers into coq. In K. V. Majumdar R., editor, *Computer Aided Verification. CAV 2017. Lecture Notes in Computer Science*, volume vol 10427. Springer, Cham, 2017.

[42] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2014.

[43] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608, 2015.

[44] R. Hasker. *The Replay of Program Derivations*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[45] D. e. a. Iain J. Whiteside. Towards formal proof script refactoring. In *18th Symposium, Calculemus 2011, and 10th International Conference, MKM*, volume LNCS 6824, pages 260–275, 2011.

[46] Y. B. Janet Bertot. Ctcoq: A system presentation. In *CADE*, pages 231–234, 1996.

[47] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, Urbana-Champaign, 1980.

[48] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. 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.

[49] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723, 2017.

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

[51] H. Li, F. Brenger, B.-Y. E. Chang, and X. Rival. Semantic-directed clumping of disjunctive abstract states. In *POPL*, page 13 pages, 2017.

[52] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.

[53] G. Malecha, A. Chlipala, and T. Braibant. Compositional computational reflection. In *ITP*, pages 374–389, 2014.

[54] G. Malecha and G. Morrisett. Mechanized verification with sharing. In *ICTAC*, 2010.

[55] G. Malecha, G. Morrisett, A. Shinnar, and R. 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.

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

[57] N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. 2006.

[58] The Coq development team. *The Coq proof assistant reference manual*, 2016.

[59] A. 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.

[60] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate

language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, March 2002.

[61] T. Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.

[62] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science*, volume vol. 2283. Springer, 2002.

[63] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat: A verified modern SAT solver. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 363–378, 2012.

[64] H. Y. 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.

[65] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hritcu, V. Sjoberg, and B. Yorgey. Software foundations. `https://www.cis.upenn.edu/~bcpierce/sf/current/index.html`.

[66] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjberg, and B. Yorgey. *Software Foundations*. 2011.

[67] C. Pit-Claudel and P. Courtieu. Company-coq: Taking proof general one step closer to a real ide. In *Coq PL*.

[68] O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *UITP*, 1998.

[69] P. R., W. T., and Z. D. Automating separation logic using smt. In S. N. and V. H., editors, *Computer Aided Verification. CAV. Lecture Notes in Computer Science*, volume 8044. Springer, Berlin, Heidelberg, 2013.

[70] U. S. Reddy. Rewriting techniques for program synthesis. *Lecture Notes in Computer Science*, 355:388–403, 1989.

[71] U. S. Reddy. Formal methods in transformational derivation of programs. In *Proc. ACM International Workshop on Formal Methods in Software Development*, Napa, CA, 1990. ACM.

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

[73] K. Roe. The heuristic theorem prover: Yet another smt-modulo theorem prover. In *Int. Conf. on Computer-Aided Verification*. Springer, 2006.

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

[75] M. Staples, D. R. Jeffery, J. Andronick, T. C. Murray, G. Klein, and R. Kolanski. Productivity for proof engineering. In *ESEM*, pages 15:1–15:4, 2014.

[76] Y.-K. T. and B. T. Automated verification of concurrent linked lists with counters. In H. M.V. and P. G., editors, *Static Analysis. SAS. Lecture Notes in Computer Science*, volume 2477, 2002.

[77] D. Tang, Y. Yu, D. Ranjan, , 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.

[78] C. Tankink. PIDE for asynchronous interation with Coq. `http://arxiv.org/pdf/1410.8221.pdf`.

[79] C. Tankink. Proof in context – web editing with rich modeless contextual feedback. In *10th International Workshop On User Interfaces for Theorem Provers*, pages 42–56, 2012.

[80] C. Tankink, H. Geuvers, J. McKinna, and F. Wiedijk. Proviola: A tool for proof re-animation. In *9th International Conference on Mathematical Knowledge Management*, 2010.

[81] M. Vijayaraghavan, A. Chlipala, A. Arvind, and N. Dave. Modular deductive

verification of multiprocessor hardware designs. In *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[82] M. Wenzel. Asynchronous user interaction and tool integration in isabelle/pide. In *Fifth International Conference, ITP*, 2014.

[83] M. Wenzel. Isabelle/jedit. `http://isabelle.in.tum.de/dist/doc/jedit.pdf`, 2014. Accessed: 2015-03-19.

[84] L. T. Yves Bertot, Gilles Kahn. Proof by pointing. In *TACS*, pages 141–160, 1994.

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

Vita

Kenneth David Roe Received his B.S. in Computer Science from the University of Michigan and Masters Degree in Computer Science from the University of Illinois. He has been running a mobile development business, Roe Mobile Development, LLC which markets the iOS app Smart Recorder which has been installed on over 1,000,000 iOS devices. Prior to joining the PhD program at Johns Hopkins, Ken spent many years working in the industry as a software engineer in many different environments. Ken's research interests are in the area of formal verification. Much of the direction of his research is based on his beliefs of what is needed to create a useful tool for industry. He has been working with the Coq theorem prover to develop practical tools for verifying software. Much of his motivation for doing formal verification research is based on his belief that formal methods could lead to much better tools for software verification in the industry.

VITA

Ken works in the Johns Hopkins Medical School as a software engineer. There he is involved in many projects aimed at improving infrastructure for doing research with large medical record databases.