Practical Tactics for Separation Logic

Andrew McCreight

Portland State University
mccreigh@cs.pdx.edu

Abstract. We present a comprehensive set of tactics that make it practical to use separation logic in a proof assistant. These tactics enable the verification of partial correctness properties of complex pointer-intensive programs. Our goal is to make separation logic as easy to use as the standard logic of a proof assistant. We have developed tactics for the simplification, rearranging, splitting, matching and rewriting of separation logic assertions as well as the discharging of a program verification condition using a separation logic description of the machine state. We have implemented our tactics in the Coq proof assistant, applying them to a deep embedding of Cminor, a C-like intermediate language used by Leroy's verified CompCert compiler. We have used our tactics to verify the safety and completeness of a Cheney copying garbage collector written in Cminor. Our ideas should be applicable to other substructural logics and imperative languages.

1 Introduction

Separation logic [1] is an extension of Hoare logic for reasoning about shared mutable data structures. Separation logic specifies the contents of individual cells of memory in a manner similar to linear logic [2], avoiding problems with reasoning about aliasing in a very natural fashion. For this reason, it has been successfully applied to the verification of a number of pointer-intensive applications such as garbage collectors [3,4].

However, most work on separation logic has involved paper, rather than machine-checkable, proofs. Mechanizing a proof can increase our confidence in the proof and potentially automate away some of the tedium in its construction. We would like to use separation logic in a proof assistant to verify deep properties of programs that may be hard to check fully automatically. This is difficult because the standard tactics of proof assistants such as Coq [5] cannot effectively deal with the linearity properties of separation logic. In contrast, work such as Smallfoot [6] focuses on the automated verification of lightweight specifications. We discuss other related work in Sect. 7.

In this paper, we address this problem with a suite of tools for separation-logic-based program verification of complex pointer-intensive programs. These tools are intended for the interactive verification of Cminor programs [7] in the Coq proof assistant, but should be readily adaptable to similar settings. We have chosen Cminor because it can be compiled using the CompCert verified

S. Berghofer et al. (Eds.): TPHOLs 2009, LNCS 5674, pp. 343–358, 2009.

compiler [7], allowing for some properties of source programs to be carried down to executable code. We have tested the applicability of these tools by using them to verifying the safety of a Cheney garbage collector [8], as well as a number of smaller examples.

The main contributions of this paper are a comprehensive set of tactics for reasoning about separation logic assertions (including simplification, rearranging, splitting, matching and rewriting) and a program logic and accompanying set of tactics for program verification using separation logic that strongly separate reasoning about memory from more standard reasoning. Together these tactics essentially transform Coq into a proof assistant for separation logic. The tactics are implemented in a combination of direct and reflective styles. The Coq implementation is available online from `http://cs.pdx.edu/~mccreigh/ptsl/`

Our tool suite has two major components. First, we have tactics for reasoning about separation logic assertions. These are focused on easing the difficulty of working with a linear-style logic within a more conventional proof assistant. These tools enable the simplification and manipulation of separation logic hypotheses and goals, as well as the discharging of goals that on paper would be trivial. These tactics are fairly modular and should be readily adaptable to other settings, from separation logic with other memory models to embeddings of linear logic in proof assistants.

The second component of our tool set is a program logic and related tactics. The program logic relates the dynamic semantics of the program to its specification. The tactics step through a procedure one statement at a time, enabling the "programmer's intuition" to guide the "logician's intuition". At each program step, there is a separation logic-based description of the current program state. A verified verification condition generator produces a precondition given the postcondition of the statement. The tactics are able to automatically solve many such steps, and update the description of the state once the current statement has been verified. Loop and branch join point annotations must be manually specified.

Organization of the Paper. We discuss the Cminor abstract machine in Sect. 2. In Sect. 3, we discuss the standard separation logic assertions we use, and the series of tactics we have created for reasoning with them. Then we discuss our program logic and its associated tactics in Sect. 4. In Sect. 5 we show how all of these pieces come together to verify the loop body of an in-place linked list reversal program. Finally, we briefly discuss our use of the previously described tactics to verify a garbage collector in Sect. 6, then discuss related work in more detail and conclude in Sect. 7.

2 Cminor

Our program tools verify programs written in Cminor, a C-like imperative language. Cminor is an intermediate language of the CompCert [7] compiler, which is a *semantics preserving* compiler from C to PowerPC assembly, giving us a

$$
\begin{aligned}
v &::= \mathsf{Vundef} \mid \mathsf{Vword}(w) \mid \mathsf{Vptr}(a) \\
e &::= v \mid \mathtt{x} \mid [e] \mid e + e \mid e\,\mathtt{!=}e \mid \ldots \\
s &::= x := e \mid [e_1] := e_2 \mid s_1; s_2 \mid \mathsf{while}(e)\ s \mid \mathsf{return}\ e \\
\sigma &::= (m, V)
\end{aligned}
$$

Fig. 1. Cminor syntax

```
y := NULL;     // y is the part of list that has been reversed
while(x != NULL) (
  t := [x+4];  [x+4] := y;     // t is next x, store prev x
  y := x;  x := t              // advance y and x
);
return y
```

Fig. 2. In-place linked list reversal

potential path to verified machine code. We use a simplified variant of Cminor that only supports 32-bit integer and pointer values. Fig. 1 gives the syntax. We write w for 32-bit integers and a for addresses, which are always 32-bit aligned. A value v is either undefined, a 32-bit word value or a pointer. We write NULL for $\mathsf{Vword}(0)$. An expression e is either a value, a program variable, a memory load (*i.e.,* a dereference), or a standard arithmetic or logical operations such as addition or comparison. In this paper, a statement s is either a variable assignment, a store to memory, a sequence of statements, a while loop or a return. In the actual implementation, while loops are implemented in terms of more primitive control structures (loops, branches, blocks and exits to enclosing blocks) capable of encoding all structured control flow. Our implementation also has procedure calls. We omit discussion of these constructs, which are supported by all of the logics and tactics presented in this paper, for reasons of space.

A memory m of type *Mem* is a partial mapping from addresses to values, while a variable environment V is a partial mapping from Cminor variables x to values. A state σ is a memory plus a variable environment. In our implementation, a state also includes a stack pointer. We define two projections $\mathsf{mem}(\sigma)$ and $\mathsf{venv}(\sigma)$ to extract the memory and variable environment components of a state.

We have formally defined a standard small-step semantics for Cminor [9], omitted here for reasons of space. Expression evaluation $\mathsf{eval}(\sigma, e)$ evaluates expression e in the context of state σ and either returns $\mathsf{Some}(v)$ if execution succeeds or None if it fails. Execution of an expression will only fail if it accesses an undefined variable or an invalid memory location. A valid memory location has been allocated by some means, such as a function call, but not freed. All other degenerate cases (such as $\mathsf{Vword}(3) + \mathsf{Vundef}$) evaluate to $\mathsf{Some}(\mathsf{Vundef})$. Unlike in C, all address arithmetic is done in terms of bytes, so the second field of an object located at address a is located at address $a + 4$.

We will the program fragment shown in Fig. 2 in the remainder of this paper to demonstrate our suite of tactics. In-place list reversal takes an argument x

that is a linked list and reverses the linked list in place, returning a pointer to the new head of the list. A linked list structure at address a has two fields, at a and $a + 4$.

3 Separation Logic Assertions

Imperative programs often have complex data structures. To reason about these data structures, separation logic assertions [1] describe memory by treating memory cells as a linear resource. In this section, we will describe separation logic assertions and associated tactics for Cminor, but they should be applicable to other imperative languages.

Fig. 3 gives the standard definitions of the separation logic assertions we use in this paper. We write P for propositions and T for types in the underlying logic, which in our case is the Calculus of Inductive Constructions [10] (CIC). Propositions have type *Prop*. We write A and B for separation logic assertions, implemented using a shallow embedding [11]. Each separation logic assertion is a memory predicate with type $Mem \to Prop$, so we write $A\ m$ for the proposition that memory m can be described by separation logic predicate A.

The separation logic assertion *contains*, written $v \mapsto v'$, holds on a memory m if v is an address that is the only element of the domain of m and $m(v) = v'$. The empty assertion **emp** only holds on empty memory. The trivial assertion **true** holds on every memory. The modal operator $!P$ from linear logic (also adapted to separation logic by Appel [12]) holds on a memory m if the proposition P is true and m is empty. The existential $\exists x : T.\ A$ is analogous to the standard existential operator. We omit the type T when it is clear from context, and follow common practice and write $a \mapsto -$ for $\exists x.\ a \mapsto x$.

The final and most crucial separation logic operator we will be using in this paper is *separating conjunction*, written $A * B$. This holds on a memory m if m can be split into two non-overlapping memories m_1 and m_2 such that A holds on m_1 and B holds on m_2. ($m = m_1 \uplus m_2$ holds if m is equal to $m_1 \cup m_2$ and the domains of m_1 and m_2 are disjoint.) This operator is associative and commutative, and we write $(A * B * C)$ for $(A * (B * C))$. This operator is used to specify the *frame rule*, which is written as follows in conventional Hoare logic:

$$\frac{\{A\}s\{A'\}}{\{A * B\}s\{A' * B\}}$$

$$\begin{array}{ll} (v \mapsto v')\ m ::= (m = \{v \rightsquigarrow v'\}) & \mathbf{emp}\ m ::= (m = \emptyset) \\ \mathbf{true}\ m \quad ::= \mathsf{True} & (!P)\ m\ ::= P \wedge \mathbf{emp}\ m \end{array}$$

$$\begin{array}{l} (\exists x : T.\ A)\ m ::= (\exists x : T.\ A\ m) \\ (A * B)\ m \quad ::= \exists m_1, m_2.\ (m = m_1 \uplus m_2) \wedge A\ m_1 \wedge\ B\ m_2 \end{array}$$

Fig. 3. Definition of separation logic assertions

B describes parts of memory that s does not interact with. The frame rule is most commonly applied at procedure call sites. We have found that we do not need to manually instantiate the frame rule, thanks to our tactics and program logic.

We can use these basic operators in conjunction with Coq's standard facilities for inductive and recursive definitions to build assertions for more complex data structures. For instance, we can inductively define a separation logic assertion $\mathsf{llist}(v, l)$ that holds on a memory that consists entirely of a linked list with its head at v containing the values in the list l. A list l (at the logical level) is either empty (written nil) or contains an element X appended to the front of another list l (written $X :: l$).

$$\begin{aligned}\mathsf{llist}(a, nil) &::= !(a = \mathsf{NULL})\\ \mathsf{llist}(a, v :: l') &::= \exists a'.\ a \mapsto v * (a+4) \mapsto a' * \mathsf{llist}(a', l')\end{aligned}$$

From this definition and basic lemmas about separation logic assertions, we can prove a number of useful properties of linked lists. For instance, if $\mathsf{llist}(v, l)$ holds on a memory and v is not NULL then the linked list is non-empty.

We can use this predicate to define part of the loop invariant for the list reversal example given in Section 2. If the variables x and y have the value v_1 and v_2, then memory m must contain two separate, non-overlapping linked lists with values l_1 and l_2. In separation logic, this is written $(\mathsf{llist}(v_1, l_1) * \mathsf{llist}(v_2, l_2))\ m$.

3.1 Tactics

Defining the basic separation logic predicates and verifying their basic properties is not difficult, even in a mechanized setting. What can be difficult is actually constructing proofs in a proof assistant such as Coq because we are attempting to carry out linear-style reasoning in a proof assistant with a native logic that is not linear.

If A, B, C and D are regular propositions, then the proposition that $(A \wedge B \wedge C \wedge D)$ implies $(B \wedge (A \wedge D) \wedge C)$ can be easily proved in a proof assistant. The assumption and goal can be automatically decomposed into their respective components, which can in turn be easily solved.

If A, B, C and D are separation logic assertions, proving the equivalent goal, that for all m, $(A * B * C * D)\ m$ implies $(B * (A * D) * C)\ m$, is more difficult. Unfolding the definition of * from Fig. 3 and breaking down the assumption in a similar way will involve large numbers of side conditions about memory equality and disjointedness. While Marti *et al.* [13] have used this approach, it throws away the abstract reasoning of separation logic.

Instead, we follow the approach of Reynolds [1] and others and reason about separation logic assertions using basic laws like associativity and commutativity. However this is not the end of our troubles. Proving the above implication requires about four applications of associativity and commutativity lemmas. This can be done manually, but becomes tedious as assertions grow larger. In real proofs, these assertions can contain more than a dozen components.

To mitigate these problems we have constructed a variety of separation logic tactics. Our goal is to make constructing proofs about separation logic predicates no more difficult than reasoning about Coq's standard logic by making the application of basic laws about separation logic assertions as simple as possible.

Now we give an example of the combined usage of our tactics. In the initial state, we have a hypothesis H that describes the current memory, and a goal on that same memory that we wish to prove. First the rewriting tactic uses a previously proved lemma about empty linked lists to simplify the assertion:

$$\frac{H : (B * A)\ m}{(A * \mathsf{llist}(v, nil) * B')\ m} \rightarrow \frac{H : (B * A)\ m}{(A * !(v = \mathsf{NULL}) * B')\ m}$$

Next the simplification tactic extracts $!(v = \mathsf{NULL})$, creating a new subgoal $v = \mathsf{NULL}$ (not shown). Finally, a matching tactic cancels out the common parts of the hypothesis and goal, leaving a smaller proof obligation.

$$\rightarrow \frac{H : (B * A)\ m}{(A * B')\ m} \rightarrow \frac{H' : B\ m'}{B'\ m'}$$

The m' in the final proof state is fresh: it must be shown that $B\ m'$ implies $B'\ m'$ for all m'[1]. This example shows most of our tactics for separation logic assertions: simplification, rearranging, splitting, matching, and rewriting. We discuss general implementation issues, then discuss each type of tactic.

Implementation. Efficiency matters for these tools. If they are slow or produce gigantic proofs they will not be useful. Our goal is to enable the interactive development of proofs, so tactics should run quickly. Our tactics are implemented entirely in Coq's tactic language L_{tac}. To improve efficiency and reliability, some tactics are implemented reflectively [14]. Implementing a tactic reflectively means implementing it mostly in a strongly typed functional language (CIC) instead of Coq's untyped imperative tactic language. Reflective tactics can be efficient because the heavy work is done at the level of propositions, not proofs. Strong typing allows many kinds of errors in the tactics to be statically detected.

Simplification. The simplification tactic ssimpl puts a separation logic assertion into a normal form to make further manipulation simpler and to clean up assertions after other tactics have been applied. ssimpl in H simplifies the hypothesis H and ssimpl simplifies the goal. Here is an example of the simplification of a hypothesis (if a goal was being simplified, the simplification would produce three goals instead of three hypotheses):

$$\frac{H : (A * \exists x.x \mapsto v * \mathbf{emp} * \mathbf{true} * (B * \mathbf{true}) * !P)\ m}{\ldots} \rightarrow \frac{\begin{array}{l} x : \mathsf{addr} \\ H' : (A * x \mapsto v * B * \mathbf{true})\ m \\ H'' : P \end{array}}{\ldots}$$

[1] The final step could be more specific (for instance, we know that m' must be a subset of m), but this is rarely useful in practice.

The basic simplifications are as follows:

1. All separation logic existentials are eliminated or introduced with Coq's meta-level existential variables (which must eventually be instantiated).
2. All modal predicates $!P$ are removed from the assertion, and either turned into new goals or hypotheses. The tactic attempts to solve any new goals.
3. All instances of **true** are combined and moved to the right.
4. For all A, $(A * \mathbf{emp})$ and $(\mathbf{emp} * A)$ are replaced with A.
5. For all A, B and C, $((A * B) * C)$ is replaced with $(A * (B * C))$.

In addition, there are a few common cases where the simplifier can make more radical simplifications:

1. If $\mathsf{Vundef} \mapsto v$ or $0 \mapsto v$ are present anywhere in the assertion (for any value v), the entire assertion is equivalent to False, because only addresses can be in the domain of $\mapsto$.
2. If an entire goal assertion can be simplified to **true** the simplifier tactic immediately solves the goal.
3. The tactic attempts to solve a newly simplified goal by assumption.

While most of this simplification could be implemented using rewriting, we have implemented it reflectively. The reflective tactic examines the assertion and simplifies it in a single pass, instead of the multiple passes that may be required for rewriting. In informal testing, this approach was faster than rewriting.

Rearranging. Our rearranging tactic srearr allows the separating conjunction's commutativity and associativity properties to be applied in a concise and declarative fashion. This is useful because the order and association of the components of a separation logic assertion affect the behavior of the splitting and rewriting tactics, which are described later. The rearranging tactic is invoked by srearr T, where T is a tree describing the final shape of the assertion. There is an equivalent tactic for hypotheses. The shape of the tree gives the desired *association* of the assertion while the numbering of the nodes give the desired *order*. Describing the desired term without explicitly writing it in the proof script makes the tactic less brittle. As with other tactics, srearr assumes that ssimpl has been used already, and thus is of the form $(A_1 * A_2 * ... * A_n)\ m$. The tactic fails if the ordering given is not a valid permutation.

Here is an example, invoked[2] with srearr $[7, 6, [[5, 4], 3], 1, 2]$:

$$\frac{\ldots}{(A * B * C * D * E * F * G)\ m} \rightarrow \frac{\ldots}{(G * F * ((E * D) * C) * A * B)\ m}$$

Permutation and reassociation are implemented as separate passes. Permutation is implemented reflectively: the tree describing the rearrangement is flattened into a list, and the assertion is also transformed into a list. For instance, the

[2] The actual syntax of the command is slightly different, to avoid colliding with existing syntactic definitions: `srearr .[7, 6, .[ .[5, 4], 3], 1, 2]%SL`. Coq's ability to define new syntax along with its implicit coercions makes this lighter weight than it would be otherwise.

tree $[[3,1],2]$ would become the list $[3,1,2]$ and the assertion $(A * B * C)$ would become the list $[A,B,C]$. The permutation list $[3,1,2]$ is used to reorder the assertion list $[A,B,C]$. In this example, the resulting assertion list is $[C,A,B]$. The initial assertion list is logically equivalent to the final one if the permutation list is a permutation of the indices of the assertion list. This requirement is dynamically checked by the tactic. Reassociation is implemented directly by examining the shape of the tree.

Splitting. The tactic ssplit subdivides a separation logic proof by creating a new subgoal for each corresponding part of the hypothesis and goal. This uses the standard separation logic property that if $(\forall m.\ A\ m \to A'\ m)$ and $(\forall m.\ B\ m \to B'\ m)$, then $(\forall m.\ (A * B)\ m \to (A' * B')\ m)$. Here is an example of the basic use of this tactic:

$$\frac{H : (A * B * C)\ m}{(E * F * G)\ m} \quad \longrightarrow \quad \frac{H_1 : A\ m_1}{E\ m_1} \quad \frac{H_2 : B\ m_2}{F\ m_2} \quad \frac{H_3 : C\ m_3}{G\ m_3}$$

Initially there is a goal and a hypothesis H describing memory. Afterward, there are three goals, each with a hypothesis. Memories m_1, m_2 and m_3 are freshly created, and represent disjoint subsets covering the original memory m. Splitting must be done with care as it can lead to a proof state where no further progress is possible.

The splitting tactic also has a number of special cases to solve subgoals involving $\mapsto$, and applies heuristics to try to solve address equalities that are generated. Here are two examples of this:

$$\frac{H : (a \mapsto v)\ m}{(a \mapsto v')\ m} \longrightarrow \frac{}{v = v'} \qquad \frac{H : (a \mapsto v)\ m}{(b \mapsto v)\ m} \longrightarrow \frac{}{a = b}$$

Matching. The matching tactic searchMatch cancels out matching parts of the hypothesis and goal. This matching is just syntactic equality, with the addition of some special cases for $\mapsto$. Here is an example of this tactic:

$$\frac{\begin{array}{l}E : v_1 = v_1' \\ H : (D * A * a \mapsto v_1 * B * b \mapsto v_2 * \mathbf{true})\ m\end{array}}{(B' * b \mapsto - * D * a \mapsto v_1' * A * \mathbf{true})\ m} \longrightarrow \frac{\begin{array}{l}E : v_1 = v_1' \\ H : (B * \mathbf{true})\ m'\end{array}}{(B' * \mathbf{true})\ m'}$$

The assertion for address a is cancelled due to the equality $v_1 = v_1'$. Notice that the predicate **true**, present in both the hypothesis and goal, is *not* cancelled. If B implies $(B' * \mathbf{true})$ then cancelling out **true** from goal and hypothesis will cause a provable goal to become unprovable. This is the same problem presented by the additive unit $\top$ in linear logic, which can consume any set of linear resources. We do not have this problem for matching other separation logic predicates as they generally do not have this sort of slack.

Matching is implemented by iterating over the assertions in the goal and hypothesis, looking for matches. Any matches that are found are placed in corresponding positions in the assertions, allowing the splitting tactic to carry out the actual cancellation.

Rewriting. In Coq, supporting rewriting of logically equivalent assertions must be implemented using the setoid rewriting facility. We have done this for the assertions described in this paper. By adding rewrite rules to a particular database, the user can extend our tools to support simplification of their own assertions.

At the beginning of Sect. 3.1, we gave an example of the use of the rewriting tactic. In the first proof state, the goal that contains an empty linked list that must be eliminated. Assume we have proved a theorem *arrayEmpty* having type $\forall x.\ \mathsf{array}(x, nil) \Rightarrow \mathbf{emp}$. The tactic srewrite *arrayEmpty* will change the proof state to the second proof state.

4 Program Logic

Our Cminor program logic is a verified verification condition generator (VCG). We discuss our VCG then the related tactics.

4.1 Verification Condition Generator

The VCG, stmPre, is a weakest precondition generator defined as a recursive function in Coq's logic. It takes as arguments the statement to be verified along with specifications for the various ways to exit the statement, and returns a state predicate that is the precondition for the statement. Verification requires showing that a user-specified precondition is as least as strong as the VC.

The design of the VCG is based on Appel and Blazy's program logic for Cminor [9]. Their program logic is structured as a traditional Hoare triple (though with more than three components), directly incorporates separation logic, and has more side conditions for some rules. On the other hand, our VCG is defined in terms of the operational semantics of the machine.

For the subset of Cminor we have defined in Sect. 2, the VCG only needs one specification argument, a state predicate q that is the postcondition of the current statement. The full version of the VCG that we have implemented takes other arguments giving the specifications of function calls, the precondition of program points that can be jumped to, and the post condition of the current procedure.

The definition of some of the cases of the VCG is given in Fig. 4. To simplify the presentation, only arguments required for these cases are included, leaving three arguments to stmPre: the statement, its postcondition q, and the current state σ. The precondition of a sequence of statements is simply the composition of VCs for the two statements: we generate a precondition for s', then use that as the postcondition of s.

For more complex statements, we have to handle the possibility of execution failing. To do this in a readable way, we use a Haskell-style do-notation to encode a sort of error monad. Operations that can fail, such as expression evaluation, return $\mathsf{Some}(R)$ if they succeed with result R, and None if they fail. Syntactically, the term "do $x \leftarrow M$; N" is similar to a let expression "let $x = M$ in N": the variable x is given the value of M, and is bound in the body N. However, the

$$\mathsf{stmPre}\ (s; s')\ q\ \sigma ::= \mathsf{stmPre}\ s\ (\mathsf{stmPre}\ s'\ q)\ \sigma$$

$$\mathsf{stmPre}\ (x := e)\ q\ \sigma ::= \mathsf{do}\ v \leftarrow \mathsf{eval}(\sigma, e);\ \mathsf{do}\ \sigma' \leftarrow \mathsf{setVar}(\sigma, x, v);\ q\ \sigma'$$

$$\mathsf{stmPre}\ ([e_1] := e_2)\ q\ \sigma ::= \mathsf{do}\ v_1 \leftarrow \mathsf{eval}(\sigma, e_1);\ \mathsf{do}\ v_2 \leftarrow \mathsf{eval}(\sigma, e_2);\ \mathsf{do}\ \sigma' \leftarrow \mathsf{storeVal}\ \sigma\ v_1\ v_2;\ q\ \sigma'$$

$$\mathsf{stmPre}\ (\mathsf{while}(e)\ s)\ q\ \sigma ::= \exists I.\ I\ \sigma \wedge \forall \sigma'.\ I\ \sigma' \rightarrow \mathsf{do}\ v \leftarrow \mathsf{eval}(\sigma', e);\ (v \neq \mathsf{Vundef}) \wedge (trueVal(v) \rightarrow \mathsf{stmPre}\ s\ I\ \sigma') \wedge (v = \mathsf{NULL} \rightarrow q\ \sigma')$$

$$(\mathsf{do}\ x \leftarrow \mathsf{Some}(v);\ P) ::= P[v/x] \qquad (\mathsf{do}\ x \leftarrow \mathsf{None};\ P) ::= \mathsf{False}$$

Fig. 4. Verification condition generator

reduction of this term differs, as shown in Fig. 4. In the case where $M = \mathsf{None}$, evaluation has failed, so the entire VC becomes equivalent to False, because failure is not allowed.

The cases for variable assignment and storing to memory follow the dynamic semantics of the machine. Variable assignment attempts to evaluate the expression e, then attempts to update the value of the variable `x`. If it succeeds, then the postcondition q must hold on the resulting state σ'. Store works in the same way: evaluation of the two expressions and a store are attempted. For the store to succeed, v_1 must be a valid address in memory. As with assignment, the postcondition q must hold on the resulting state. With both of these statements, if any of the intermediate evaluations fail then the entire VC will end up being False, and thus impossible to prove.

The case for while loops is fairly standard. A state predicate I must be selected as a loop invariant. I must hold on the initial state σ. Furthermore, for any other states σ' such that $I\ \sigma'$ holds, it must be possible to evaluate the expression e to a value v, which cannot be Vundef. If the value v is a "true" value (*i.e.,* is either a pointer or a non-zero word value) then the precondition of the loop body s must hold, where the postcondition of the body is I. If the value is false (equal to $\mathsf{Vword}(0)$) then the postcondition of the entire loop must hold.

We have mechanically verified the soundness of the verification condition generator as part of the safety of the program logic: if the program is well-formed, then we can either take another step or we have reached a valid termination state for the program.

4.2 Variable Environment Reasoning

We use separation logic to reason about memory, but define a new predicate to reason about variable environments. At any given point in a program, the contents of the variable environment V can be described by a predicate $\mathsf{veEqv}\ S\ V'\ V$. S is a set of variables that are valid in V, and V' gives the values of some of the variables in V.

4.3 Tactics

The tactic vcSteps lazily unfold the VC and attempts to use the separation logic description of the state to perform symbolic execution to step through the VC. Fig. 5 shows the rough sequence of steps that vcSteps carries out automatically at an assignment. Each numbered line below the horizontal line gives an intermediate goal state as vcSteps is running. The two hypothesis V and H above the line describe the variable environment and memory of the initial state σ, and are the precondition of this statement. The first stmPre below the line is the initial VC that must be verified. The tactic unfolds the definition of the VC for a sequence (line 2), then for an assignment (line 3), then determines that the value of the variable `x` is a by examining the hypothesis V (line 4). The load expression now has a value as an argument, so the tactic will examine the hypothesis H to determine that address a contains value v (line 5). The relevant binding $a \mapsto v$ can occur as any subtree of the separation logic assertion. Now the do-notation can be reduced away (line 6).

Once this is done, all that remains is to actually perform the assignment. The hypothesis V proves that `y` is in the domain the variable file of σ, so setting the value of `y` to v will succeed, producing a new state σ'. The tactic simplifies the goal to step through this update, and uses V to produce a new V' that describes σ'. H can still be used as the memory of σ' is the same as the memory of σ. This results in the proof state shown in Fig. 6. The tactic can now begin to analyze the statement s in a similar manner.

This may seem like a lengthy series of steps, but it is largely invisible to the user. Breaking down statements in this manner allows the tactics to easily handle a wide variety of expressions. vcSteps will get "stuck" at various points that require user intervention, such as loops and branches where invariants must be supplied, and where the tactic cannot easily show that a memory or variable

$$\frac{\begin{array}{l} V : \mathsf{veEqv}\ \{\mathtt{x},\mathtt{y}\}\ \{(\mathtt{x} \rightsquigarrow a)\}\ (\mathsf{venv}(\sigma)) \\ H : (A * a \mapsto v * B)\ (\mathsf{mem}(\sigma)) \end{array}}{\begin{array}{l} 1)\ \mathsf{stmPre}\ (\mathtt{y} := [\mathtt{x}];\ s)\ P\ \sigma \\ 2)\ \mathsf{stmPre}\ (\mathtt{y} := [\mathtt{x}])\ (\mathsf{stmPre}\ s\ P)\ \sigma \\ 3)\ \mathsf{do}\ v' \leftarrow \mathsf{eval}(\sigma, [\mathtt{x}]);\ \mathsf{do}\ \sigma' \leftarrow \mathsf{setVar}(\sigma, \mathtt{y}, v');\ \ \mathsf{stmPre}\ s\ P\ \sigma' \\ 4)\ \mathsf{do}\ v' \leftarrow \mathsf{eval}(\sigma, [a]);\ \mathsf{do}\ \sigma' \leftarrow \mathsf{setVar}(\sigma, \mathtt{y}, v');\ \ \mathsf{stmPre}\ s\ P\ \sigma' \\ 5)\ \mathsf{do}\ v' \leftarrow \mathsf{Some}(v);\ \mathsf{do}\ \sigma' \leftarrow \mathsf{setVar}(\sigma, \mathtt{y}, v');\ \ \mathsf{stmPre}\ s\ P\ \sigma' \\ 6)\ \mathsf{do}\ \sigma' \leftarrow \mathsf{setVar}(\sigma, \mathtt{y}, v);\ \ \mathsf{stmPre}\ s\ P\ \sigma' \end{array}}$$

Fig. 5. Program logic tactics: examining the state

$$\frac{\begin{array}{l} V' : \mathsf{veEqv}\ \{\mathtt{x},\mathtt{y}\}\ \{(\mathtt{x} \rightsquigarrow a), (\mathtt{y} \rightsquigarrow v)\}\ (\mathsf{venv}(\sigma')) \\ H : (A * a \mapsto v * B)\ (\mathsf{mem}(\sigma')) \end{array}}{\mathsf{stmPre}\ s\ P\ \sigma'}$$

Fig. 6. Program logic tactics: updating the state

operation is safe. In the latter case, the tactics described in the previous section can be applied to manipulate the assertion to a form the program logic tactics can understand, then vcSteps can be invoked again to pick up where it left off.

In addition to the tactic for reasoning about VCs, there is a tactic veEqvSolver to automatically solve goals involving veEqv. This is straightforward, as it only needs to reason about concrete finite sets.

5 Example of Tactic Use

In this section, we demonstrate the use of our tactics by verifying a fragment of an in-place linked list reversal, given in Fig. 2. Before we can verify this program, we need to define a loop invariant $inv\ l_0\ \sigma$, where l_0 is the list of values in the initial linked list and σ is the state at a loop entry:

$$\begin{aligned} inv\ l_0\ \sigma ::= {} & \exists v_1, v_2.\ \mathsf{veEqv}\ \{\mathtt{x}, \mathtt{y}, \mathtt{t}\}\ \{(\mathtt{x}, v_1), (\mathtt{y}, v_2)\}\ (\mathsf{venv}(\sigma)) \wedge \\ & \exists l_1, l_2.\ (\mathsf{llist}(v_1, l_1) * \mathsf{llist}(v_2, l_2))\ (\mathsf{mem}(\sigma)) \wedge \\ & \mathsf{rev}(l_1) +\!\!+ l_2 = \mathsf{rev}(l_0) \end{aligned}$$

In the first line, the predicate veEqv requires that in the current state that at least the program variables `x`, `y`, `t` are valid, and that the variables `x` and `y` are equal to some values v_1 and v_2, respectively. The second line is a separation logic predicate specifying that memory contains two disjoint linked lists as seen in Sect. 3. From these two descriptions, we can deduce that the variable `x` contains a pointer to a linked list containing the values l_1. Finally, the invariant requires that reversing l_1 and appending l_2 results in the original list l_0. We write $\mathsf{rev}(l)$ for the reversal of list l and $l +\!\!+ l'$ for appending list l' to the end of l.

To save space, we will only go over the verification of the loop body and not describe in detail the invocation of standard Coq tactics. In the loop body, we know that the loop invariant inv holds on the current state and that the value of `x` is not $\mathsf{Vword}(0)$ (*i.e.,* NULL). We must show that after the loop body has executed that the loop invariant is reestablished.

Our initial proof state is thus:

$$\frac{\begin{array}{l} NE : v_1 \neq \quad Vint0 \\ L : \mathsf{rev}(l_1) +\!\!+ l_2 = \mathsf{rev}(l_0) \\ H : (\mathsf{llist}(v_1, l_1) * \mathsf{llist}(v_2, l_2))\ (\mathsf{mem}(\sigma)) \\ V : \mathsf{veEqv}\ \{\mathtt{x}, \mathtt{y}, \mathtt{t}\}\ \{(\mathtt{x}, v_1), (\mathtt{y}, v_2)\}\ (\mathsf{venv}(\sigma)) \end{array}}{\mathsf{stmPre}\ (\mathtt{t} := [\mathtt{x} + 4]; [\mathtt{x} + 4] := \mathtt{y}; \mathtt{y} := \mathtt{x}; \mathtt{x} := \mathtt{t})\ (inv\ l_0)\ \sigma}$$

Our database of rewriting rules for separation logic data structures includes the following rule: if v is not 0, then a linked list $\mathsf{llist}(v, l)$ must have at least one element. Thus, applying our rewriting tactic to the hypothesis H triggers this rule for $v = v_1$. After applying a standard substitution tactic, we have this proof state (where everything that is unchanged is left as ...):

$$\frac{\begin{array}{l}\ldots\\ L : \mathsf{rev}(v :: l_1') +\!\!+ l_2 = \mathsf{rev}(l_0)\\ H' : ((v_1 \mapsto v) * (v_1{+}4 \mapsto v_1') * \mathsf{llist}(v_1', l_1') * \mathsf{llist}(v_2, l_2))\ (\mathsf{mem}(\sigma))\end{array}}{\ldots}$$

Now that we know that the address $v_1 + 4$ contains the value v_1', we can show that it is safe to execute the loop body. The tactic `vcSteps`, described in Sect. 4.3, is able to automatically step through the entire loop body, leaving an updated state description and the goal of showing that the loop invariant holds on the final state of the loop σ':

$$\frac{\begin{array}{l}\ldots\\ H'' : ((v_1 \mapsto v) * (v_1{+}4 \mapsto v_2) * \mathsf{llist}(v_1', l_1') * \mathsf{llist}(v_2, l_2))\ (\mathsf{mem}(\sigma'))\\ V' : \mathsf{veEqv}\ \{\mathtt{x}, \mathtt{y}, \mathtt{t}\}\ \{(\mathtt{x}, v_1'), (\mathtt{y}, v_1), (\mathtt{t}, v_1')\}\ (\mathsf{venv}(\sigma'))\end{array}}{inv\ l_0\ \sigma'}$$

Now we must instantiate two existential variables and show that they are the values of the variables `x` and `y` (and that `x`, `y` and `t` are valid variables). These existentials can be automatically instantiated, and the part of the goal using veEqv solved, using a few standard Coq tactics along with the veEqvSolver described in Sect. 4.3.

The remaining goal is

$$\frac{\ldots}{\exists l_3, l_4.\ (\mathsf{llist}(v_1', l_3) * \mathsf{llist}(v_1, l_4))\ (\mathsf{mem}(\sigma'))\ \wedge\ \mathsf{rev}(l_3) +\!\!+ l_4 = \mathsf{rev}(l_0)}$$

We manually instantiate the existentials with l_1' and $(v :: l_2)$ and split the resulting conjunction using standard Coq tactics. This produces two subgoals. The second subgoal is $\mathsf{rev}(l_2) +\!\!+ (v :: l_1') = \mathsf{rev}(l_0)$ and can be solved using standard tactics. The first subgoal is an assertion containing $\mathsf{llist}(v_1', v :: l_2)$, which we can be simplified using standard tactics leaving the proof state

$$\frac{\ldots}{(\mathsf{llist}(v_1', l_1') * (\exists x'.(v_1 \mapsto v) * (v_1{+}4 \mapsto x') * \mathsf{llist}(x', l_2)))\ (\mathsf{mem}(\sigma'))}$$

Invoking our simplification tactic `ssimpl` replaces the existential with a Coq meta-level existential variable "?100", leaving the goal[3]

$$\frac{\begin{array}{l}\ldots\\ H'' : ((v_1 \mapsto v) * (v_1{+}4 \mapsto v_2) * \mathsf{llist}(v_1', l_1') * \mathsf{llist}(v_2, l_2))\ (\mathsf{mem}(\sigma'))\end{array}}{(\mathsf{llist}(v_1', l_1') * (v_1 \mapsto v) * (v_1{+}4 \mapsto ?100) * \mathsf{llist}(?100, l_2))\ (\mathsf{mem}(\sigma'))}$$

This goal is immediately solved by our tactic `searchMatch`. The hypothesis contains $v_1{+}4 \mapsto v_2$, and the goal contains $v_1{+}4 \mapsto ?100$, so ?100 must be equal to v_2. Once ?100 is instantiated, the rest of easy to match up. Thus we have verified that the body of the loop preserves the loop invariant.

[3] H'' has not changed but we include it here for convenience.

6 Implementation and Application

Our tactics are implemented entirely in the Coq tactic language L_{tac}. The tool suite include about 5200 lines of libraries, such as theorems about modular arithmetic and data structures such as finite sets. Our definition of Cminor (discussed in Sect. 2) is about 4700 lines. The definition of separation logic assertions and associated lemmas (discussed in Sect. 3) are about 1100 lines, while the tactics are about 3000 lines. Finally, the program logic (discussed in Sect. 4), which includes its definition, proofs, and associated tactics, is about 2000 lines.

We have used the tactics described in this paper to verify the safety and completeness of a Cheney copying garbage collector [8] implemented in Cminor. It is *safe* because the final object heap is isomorphic to the reachable objects in the initial state and *complete* because it only copies reachable objects. This collector supports features such as scanning roots stored in stack frames, objects with an arbitrary number of fields, and precise collection via information stored in object headers. The verification took around 4700 lines of proof scripts (including the definition of the collector and all specifications), compared to 7800 lines for our previous work [4] which used more primitive tactics. The reduction in line count is despite the fact that our earlier collector did not support any of the features we listed earlier in this paragraph and did not use modular arithmetic.

There has been other work on mechanized garbage collector verification, such as Myreen *et al.* [15] who verify a Cheney collector in 2000 lines using a decompilation based approach. That publication unfortunately does not have enough detail of the collector verification to explain the difference in proof size, though it is likely due in part to greater automation. Hawblitzel *et al.* [16] used a theorem prover to automatically verify a collector that is realistic enough to be used for real C# benchmarks.

7 Related Work and Conclusion

Appel's unpublished note [12] describes Coq tactics that are very similar to ours. These tactics do not support as many direct manipulations of assertions, and his rewriting tactic appears to require manual instantiation of quantifiers. The paper describes a tactic for inversion of inductively defined separation logic predicates, which we do not support. While Appel also applies a "two-level approach" that attempts to pull things out of separation logic assertions to leverage existing proof assistant infrastructure, our approach is more aggressive about this, lifting out expression evaluation. This allows our approach to avoid reasoning about whether expressions in assertions involve memory.

We can give a rough comparison of proof sizes, using the in-place list reversal procedure. Ignoring comments and the definition of the program, by the count of `wc` Appel uses 200 lines and 795 words to verify this program [12]. With our tactics, ignoring comments, blank lines and the definition of the program, our verification takes 68 lines and less than 400 words.

Affeldt and Marti [13] use separation logic in a proof assistant, but unfold the definitions of the separation logic assertions to allow the use of more conventional

tactics. Tuch *et al.* [17] define a mechanized program logic for reasoning about C-like memory models. They are able to verify programs using separation logic, but do not have any complex tactics for separation logic connectives.

Other work, such as Smallfoot [6], has focused on automated verification of lightweight separation logic specifications. This approach has been used as the basis for certified separation logic decisions procedures in Coq [18] and HOL [19]. Calcagno *et al.* [20] use separation logic for an efficient compositional shape analysis that is able to infer some specifications.

Still other work has focused on mechanized reasoning about imperative pointer programs outside of the context of separation logic [11,21,22] using either deep or shallow embeddings. Expressing assertions via more conventional propositions enables the use of powerful preexisting theorem provers. Another approach to program verification decompiles imperative programs into functional programs that are more amenable to analysis in a proof assistant [23,15].

The tactics we have described in this paper provide a solid foundation for the use of separation logic in a proof assistant but there is room for further automation. Integrating a Smallfoot-like decision procedure into our tactics would automate reasoning about standard data structures.

We have presented a set of separation logic tactics that allows the verification of programs using separation logic in a proof assistant. These tactics allow Coq to be used as a proof assistant for separation logic by allowing the assertions to be easily manipulated via simplification, rearranging, splitting, matching and rewriting. They also provide tactics for proving a verification condition by means of a separation logic based description of the program state. These tactics are powerful enough to verify a garbage collector.

Acknowledgments. I would like to thank Andrew Tolmach for providing extensive feedback about the tactics, and Andrew Tolmach, Jim Hook and the anonymous reviewers for providing helpful comments on this paper.

References

1. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, Washington, DC, USA, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
2. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
3. Birkedal, L., Torp-Smith, N., Reynolds, J.C.: Local reasoning about a copying garbage collector. In: POPL 2005, pp. 220–231. ACM Press, New York (2004)
4. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying gcs and their mutators. In: PLDI 2007, pp. 468–479. ACM, New York (2007)
5. The Coq Development Team: The Coq proof assistant, `http://coq.inria.fr`
6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
7. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54. ACM Press, New York (2006)

8. Cheney, C.J.: A nonrecursive list compacting algorithm. Communications of the ACM 13(11), 677–678 (1970)
9. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 5–21. Springer, Heidelberg (2007)
10. Paulin-Mohring, C.: Inductive definitions in the system Coq—rules and properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664. Springer, Heidelberg (1993)
11. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)
12. Appel, A.W.: Tactics for separation logic (January 2006), `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`
13. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an os using separation logic. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 400–419. Springer, Heidelberg (2006)
14. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Ito, T., Abadi, M. (eds.) TACS 1997. LNCS, vol. 1281, pp. 515–529. Springer, Heidelberg (1997)
15. Myreen, M.O., Slind, K., Gordon, M.J.C.: Machine-code verification for multiple architectures - an application of decompilation into logic. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD) (2008)
16. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL 2009, pp. 441–453. ACM, New York (2009)
17. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL 2007, pp. 97–108. ACM, New York (2007)
18. Marti, N., Affeldt, R.: A certified verifier for a fragment of separation logic. In: 9th JSSST Workshop on Programming and Prog. Langs, PPL 2007 (2007)
19. Tuerk, T.: A separation logic framework in HOL. In: Otmane Ait Mohamed, C.M., Tahar, S. (eds.) TPHOLs 2008, August 2008, pp. 116–122 (2008)
20. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL 2009, pp. 289–300. ACM, New York (2009)
21. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. 199(1-2), 200–227 (2005)
22. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
23. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)