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 to verify 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 reasons about 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] do not cannot effectively deal with the linearity properties of separation logic.

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 [6] 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 compiler [6], allowing for the possibility of 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 [7], 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 other more conventional 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 conventional proof assistants. Some of these tactics should be useful for reasoning about any kind of associative and commutative operator.

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 simplify the task of showing that the current state is safe to execute by automatically solving many such steps and updating the description of the state once the current statement has been verified. Loop and branch join point annotations must be manually specified.

The work most closely related to ours is unpublished work by Appel on tactics for separation logic [8], which also defines a set of tactics for manual proofs of separation logic in proof assistants, but has less support for general reasoning about separation logic assertions and has somewhat more manual proofs. Other work by Berdine *et al.* [9] and Calcagno *et al.* [10] is focused on the verification of lightweight properties of large code bases, rather than heavier properties for smaller code bases. We developed an earlier generation of tactics for verifying programs using separation logic [11] in Coq for a less realistic machine and used them to verify a series of garbage collectors [4].

Organization of the Paper. First we will give an overview of separation logic assertions, then discuss our tactics for proving lemmas involving separation logic assertions. After that we will discuss our program logic, and the various tactics we have created to verify programs. Then we will briefly discuss how we used our

$$v ::= \mathsf{Vundef} \mid \mathsf{Vword}(w) \mid \mathsf{Vptr}(a)$$

$$e ::= v \mid \mathtt{x} \mid [e] \mid e + e \mid ...$$

$$s ::= \mathsf{skip} \mid \mathtt{x} := e \mid [e_1] := e_2 \mid \mathsf{loop}\ s \mid s_1; s_2 \mid ...$$

$$\sigma ::= (m, v)$$

Fig. 1. Cminor syntax

separation logic tactics to verify a garbage collector, then discuss related work in more detail and conclude.

2 Example

In this section we give an example of our separation logic tactics.

[[Extended example of a proof]]

3 Cminor

Our program tools verify programs written in Cminor, a C-like imperative language. The main difference between C and Cminor is that local variables are not allocated on the stack, so it is not possible to take the address of a local variable. Cminor is an intermediate language of the CompCert [6] compiler. CompCert is a *semantics preserving* compiler from C to PowerPC assembly, giving us a path to verified machine code if we can prove properties about the programs being compiled.

We use a simplified variant of Cminor that only supports 32-bit values and does not support floating point numbers. Fig. **??** 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. An expression e is either a value, a program variable, a memory load, or any of a nymber of standard arithmetic or logical operations such as addition. A statement is either a skip s that does nothing, a variable assignment, a store to memory, an infinite loop, or a sequence of statements. In addition, there are statements for blocks, exits to blocks, procedure calls, procedure returns, and branches. Our tactics support these additional statements plus procedure definitions, but we omit discussion of them here for space reasons.

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 $\mathtt{x}$ to values. A state σ is a memory plus a variable environment. In our implementation, a state also includes a stack pointer.

We have defined a standard small-step semantics for Cminor, which is standard [16] and omitted here for reasons of space. They are formally defined in

$$
\begin{array}{ll}
(v \mapsto v')\ m & ::= (m = \{v \rightsquigarrow v'\}) \\
\mathbf{emp}\ m & ::= (m = \emptyset) \\
\mathbf{true}\ m & ::= \mathsf{True} \\
(!P)\ m & ::= P \wedge \mathbf{emp}\ m \\
(\exists x : T.\ A)\ m & ::= (\exists x : T.\ A\ m) \\
(A * B)\ m & ::= \exists m_1, m_2.\ (m = m_1 \uplus m_2) \wedge A\ m_1 \wedge\ B\ m_2
\end{array}
$$

Fig. 2. Definition of separation logic assertions

the Coq framework. 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. The only way execution of an expression can fail is if it accesses an undefined variable or an invalid memory location. All other degenerate cases (such as adding an integer to Vundef) simply return $\mathsf{Some}(\mathsf{Vundef})$.

4 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. 1 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, the Calculus of Inductive Constructions [12] (CIC). Propositions have type *Prop*. We write A and B for separation logic assertions, which are implemented using a shallow embedding [13]. Each separation logic assertion is a memory predicate with type $Mem \rightarrow Prop$, 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 $m(v) = v'$. We write such a memory m as $\{v \rightsquigarrow v'\}$. The next assertion is *empty*, written **emp**, which 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 [8]) holds on a memory m if the proposition P holds and m is empty. This is useful when defining more complex assertions and aids automated analysis by syntactically distinguishing pure propositions that do not involve memory. The existential $\exists x : T.\ A$ is similar to the standard existential operator, where x is bound in A, and holds on a memory m if there exists some term M having type T, such that A with M substituted for x holds on m. 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 $A * B.\ m = m_1 \uplus m_2$ holds if m is equal to $m_1 \cup m_2$ and the domains of m_1 and m_2 are disjoint. This 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. This operator is right 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:

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

B describes the part of memory that s does not interact with. The frame rule is most commonly applied at procedure call sites. We have found that we have never needed to manually instantiate the frame rule, thanks to a combination of 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 an assertion $\mathsf{array}(a, l)$ that holds on a memory when that memory consists entirely of an array starting at address a containing values given by the list l:

$$\mathsf{array}(a, nil) ::= \mathbf{emp}$$
$$\mathsf{array}(a, v :: l) ::= a \mapsto v * \mathsf{array}(a + 4, l)$$

4.1 Tactics

Defining the basic separation logic predicates and verifying the basic properties is not difficult to do using conventional techniques, even in a mechanized setting. What can be difficult is actually constructing proofs in a proof assistant such as Coq. As we have said, the primary difficulty is that 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. A single tactic invocation will break down the assumption and goal into their respective components and match up hypotheses to goals.

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. 1 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.* [14] 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 will require around four applications of associativity and commutativity lemmas. This can be done manually, but becomes increasingly tedious as assertions grow larger. In real proofs, these assertions can contain more than a dozen components.

$$\frac{H : (B * A)\ m}{(A * \mathsf{array}(x, nil) * B')\ m}\ \text{(initial)} \qquad \frac{H : (B * A)\ m}{(A * \mathbf{emp} * B')\ m}\ \text{(rewrite)} \qquad \frac{H : (B * A)\ m}{(A * B')\ m}\ \text{(simplify)} \qquad \frac{H' : B\ m'}{B'\ m'}\ \text{(match)}$$

Fig. 3. Example of tactic usage

To mitigate these and other problems we have constructed a variety of custom 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.

Fig. 3 contains an example of the usage of our tactics. Initially, we have a hypothesis H that is a proof that $(B * A)\ m$, and we are trying to prove that $(A{*}\mathsf{array}(x, nil){*}B')\ m$. First the rewriting tactic uses a previously proven lemma about empty arrays to clean away the array assertion. Next the simplification tactic removes the **emp**. Finally, a matching tactic cancels out the common parts of the hypothesis and goal. This leaves us with a smaller proof obligation.

We have five types of tactics for separation logic assertions: simplification, splitting, matching, rewriting and rearranging. We will discuss general implementation issues, then discuss each type of tactic in turn.

Implementation Efficiency matters for these tools. If they are slow or produce gigantic proofs they will not be useful. All of our tactics are implemented entirely in Coq's tactic language L_{tac}. To improve efficiency and reliability, some tactics are implemented reflectively [15].

The core of a reflective tactic is implemented in the proof language (in our case, CIC) as a transformation on a deep embedding of, say, separation logic. Only a thin wrapper layer is implemented in the tactic language. Reflective tactics provide a number of software engineering benefits, stemming from the fact that they allow the bulk of a tactic to be implemented in a strongly typed functional language instead of in the tactic language, which at least for Coq is untyped and imperative. Additionally, the tactic can be mechanically reasoned about, as it is implemented in the proof language, enabling verification of correctness and completeness. Reflective tactics can also be smaller than proofs implemented using rewriting, as the proof does not get larger as more transformations are carried out.

Simplification. The simplification tactic ssimpl puts a separation logic assertion into a normal form to make manipulation simpler. The primary purpose of this tactic is to clean up assertions after definitions have been unfolded or other tactics have been applied. ssimpl in H simplifies the hypothesis H and ssimpl simplifies the goal. We give an example of simplification in Fig. 4. The P on the

$$(A * \exists x.x \mapsto v * \mathbf{emp} * \mathbf{true} * (B * \mathbf{true}) * !P)\ m \qquad (A * x \mapsto v * B * \mathbf{true})\ m \quad P$$

(before) (after)

Fig. 4. Example simplification

right of the "after" part of the figure is a separate hypothesis if a hypothesis is being simplified or a separate subgoal if the goal is being simplified.

The basic simplifications are as follows:

1. All separation logic existentials are removed. In the hypothesis, they are introduced as new variables, while in the goal new meta-level existential variables are created that must eventually be instantiated.
2. In a similar fashion to existentials, all "pure" predicates $!P$ are removed from the assertion, and either turned into new goals or hypotheses, depending on what is being eliminated. The tactic attempts to solve any new goals.
3. Any instances of **true** are combined and moved to the right hand side of the assertion. This can be done because **true** is equivalent to $\mathbf{true} * \mathbf{true}$. **true** is moved to the right because it is likely to be the least interesting part of any assertion.
4. For all A, $(A * \mathbf{emp})$ and $(\mathbf{emp} * A)$ are replaced with A. **emp** is the identity element of $*$, and so can be eliminated in this way.
5. For all A, B and C, $((A * B) * C)$ is replaced with $(A * (B * C))$. Having assertions in a canonical form like this makes manipulation by other tools easier.

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. From the definition of $\mapsto$, the first value must always be an address, and Vundef and 0 are not addresses, so we have a contradiction. Other non-address values could be added, but these are two that arise frequently. If we are simplifying a hypothesis, the tactic can then immediately solve the current goal, no matter what it is.
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.

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 in Sects. 2.1 and **??**.

$$(A * B * C * D * E * F * G)\ m \qquad (G * F * ((E * D) * C) * A * B)\ m$$

(before) (after)

srearr [7, 6, [[5, 4], 3], 1, 2]

Fig. 5. Example of declarative rearranging

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

(before) (after)

Fig. 6. Example of splitting

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, srearr T in H, which performs the rearranging on hypothesis H. Fig. 8 shows an example of the use of this tactic.[1] The shape of the tree gives the desired *association* of the assertion while the nodes give the desired *order*. Describing the desired term in this abstract way (as opposed to asserting the desired final term in its entirety and using searchMatch) makes the tactic argument less brittle in the face of minor changes. There are other simpler tactics that can be used when a less radical restructuring is needed.

For the association, notice that the placement of parenthesis in the final state matches the placement of the parenthesis in the tactic argument. For the ordering, the number at a given position is the initial position of that term. In the example, 7 is the first leaf in argument to the tactic, so the first part of the final assertion is G, the seventh part of the initial assertion. 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 will fail if the ordering given is not a valid permutation.

Splitting. The tactic ssplit subdivides a separation logic proof by creating a new subgoal for each corresponding part of the hypothesis and goal. Fig. 5 gives an example of ssplit. Initially there is a single goal and a single hypothesis describing memory. Afterwards, 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. If all three implications hold, then the initial implication also holds.

[1] 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 provides a fairly lightweight notation.

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

(before) (after) (before) (after) (before) (after)

Fig. 7. Special cases for contains

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

(before) (after)

Fig. 8. Matching example

Splitting must be done with care as it can lead to a proof state where no further progress is possible. In Fig. 5, if A does not imply E, then the "after" state cannot be proven, even if the "before" state could (if, say, A implied F, B implied E, and C implied G).

The splitting tactic also has a number of special cases to solve subgoals involving $\mapsto$, shown in Fig. 6. For the first case, if a memory location a is known to contain a value v and the goal is to show that a contains v', then we can simplify this to showing that $v = v'$. For the second case, recall that $a \mapsto -$ is defined to be $\exists x.\ a \mapsto x$. Instantiating the existential with v is enough to solve the goal. For the third case, this implication will be true if and only if the addresses a and b are equal. The tactic also applies some heuristics to try to solve the address equality. For instance, $a + 0$ can be simplified to a, and $a + b = a + c$ can be simplified to $b = c$. The splitting tactic also handles any combination of these three special cases. For instance, if neither the address nor the value match between hypothesis and goal, then two equality subgoals are created. These special cases are not remarkable, but the automatic simplification provided by the splitting tactic eliminates many tedious proof steps.

Matching. The matching tactic searchMatch cancels out commonalities between the hypothesis and goal. It also implements the first two special cases for $\mapsto$ given in Fig. 6. As with the other tools, it assumes that ssimpl has already been run on the goal and hypothesis.

Fig. 7 gives an example of the effect of the matching tactic. In the initial state, we have both a description of memory and a proof that the value v_1 is equal to the value v_1'. After the tactic runs, assertions such as A and D that are present in both the top and bottom are removed. In addition, the two $\mapsto$ assertions are also automatically cancelled out. The assertion for address a can be cancelled out due to the equality $v_1 = v_1'$, while the assertion for address b can be cancelled by instantiation of the existential in $b \mapsto -$.

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. **true** behaves like an elastic garbage bag into which we can stuff as little or as much of the memory as we wish, but we cannot tell what we will need to place in it until the rest of the assertion has been accounted for. 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.

Rewriting. Most of our tactics allow the standard connectives and formulae of separation logic such as $*, \mapsto$ and **emp** to be easily manipulated, but real programs will also require assertions about higher-level data structures such as arrays (see Sect. 2), linked lists, and even more complex application-specific data structures such as garbage collector heaps. We want to support the lightweight manipulation of new assertions in a manner that is easily extended by the user. We solve this problem by implementing a form of rewriting for separation logic assertions. The setoid rewriting tactics of Coq allow user-defined relations to be used with the standard rewriting tactics. For separation logic assertions, there are two relations of interest: assertion implication (written $A \Rightarrow B$) and equivalence (written $A \Leftrightarrow B$). These are defined in terms of implication and logical equivalence: $A \Rightarrow B$ is defined as $\forall m.\ A\ m \rightarrow B\ m$, while $A \Leftrightarrow B$ is defined as $\forall m.\ A\ m \leftrightarrow B\ m$.

In Fig. 3 we gave an example of the use of the rewriting tactic. In the first proof state, the goal that contains an empty array that must be eliminated. Assume we have proved a theorem *arrayEmpty* having type $\forall x.\ \mathsf{array}(x, nil) \Rightarrow$ **emp**. The tactic srewrite *arrayEmpty* will change the proof state to the state shown in the second part of the figure. Manual instantiation of the quantifier x is not required because srewrite is a thin wrapper[2] around Coq's standard rewrite tactic, which uses unification to instantiate quantifiers in the types of terms.

Once basic rewriting is set up for separation logic assertions, we can take advantage Coq's autorewrite tactic that will repeatedly apply a user-defined database of rewriting rules. A user can define a database of simplification rules for their application-specific data structure assertions, then use autorewrite followed by ssimpl to create a new simplification tactic that is capable of handling the new assertions.

5 Program Logic

Our Cminor program logic, CMPL, is a verified verification condition generator (VCG). We will discuss our VC then describe how verification conditions (VCs) are verified.

[2] The setoid rewriting facility requires a syntactic hint as to which equivalence to use, so srewrite changes a goal of the form $A\ m$ to mpOk $A\ m$ before applying the standard rewriting tactic, where mpOk $A\ m$ is defined to be $A\ m$.

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

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

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

$$\mathsf{stmPre}\ (\texttt{loop}\ s)\ q\ \sigma ::= \exists I.\ I\ \sigma \wedge (\forall \sigma'.\ I\ \sigma' \rightarrow \mathsf{stmPre}\ s\ I\ \sigma')$$

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

Fig. 9. Verification condition generator

5.1 Verification Condition Generator

The VCG, stmPre, is a weakest precondition generator, defined as a recursive function in Coq's logic, and 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. Verifying the VC then requires showing that a user-specified precondition is as least as strong as the VC-generated precondition.

The design of the VCG is based on Appel and Blazy's program logic for Cminor[16]. Their logic is structured as a traditional Hoare "triple" (though there are more than three components) whereas our logic is more like a weakest precondition generator. Their logic builds in separation logic instead of being defined in terms of the operational semantics of the machine, and has more side conditions for various rules (for instance, their logic only allows pure expressions in store statements).

For the subset of Cminor we have defined in Sect. **??**, 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 blocks that can be exited to, and the post condition of the current procedure.

The VC does not contain any separation logic assertions. Instead, it is defined in terms of the operational semantics of the Cminor abstract machine. The program logic tactics relate separation logic assertions to the operational semantics, allowing iterative refinement of the program logic tactics without changing the program logic itself.

The definition of some of the cases of the VCG is given in Fig. 9. Only arguments that required for these cases are included to simplify the presentation, leaving three arguments to stmPre: the statement we are generating a precondition for, the precondition of the following statement q, and the current state σ. Note that the preconditions are *state* predicates (involving memory and a variable environment) while the separation logic assertions defined in previous sections are *memory* predicates.

A skip statement always succeeds and passes to the next statement, so the precondition is just q. 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. The operations that can fail, such as expression evaluation, return $\mathsf{Some}(R)$ if they succeed and produce some result R, and None if they fail. If the operation succeeds, the returned value is substituted for the variable in the body of the do, P, which must be a proposition. If the operation fails, the do-notation reduces to the predicate False.

The cases for variable assignment and storing to memory follow the dynamic semantics of the machine. Variable assignment attempts to evaluate the right hand side of the statement e, then attempts to update the value of the variable `x` in the initial state. The latter will fail if `x` is not declared in the current function. If it does succeed, then the postcondition q must hold on the resulting state σ'. Store works in the same way: evaluation of the two expressions, then 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 the VC will be impossible to prove.

Finally, the case for loops is standard. A state predicate I must be selected as a loop invariant. The loop invariant must hold on the initial state, and the loop invariant must imply the precondition of the loop body s, when the postcondition is I.

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.

5.2 Tactics

The VC generator described in the previous section produces large VCs that do not mention separation logic assertions. To manage the size and allow separation logic based reasoning, we have created a tactic $\mathsf{vcSteps}$ that examines and updates a description of the program state for the statement that is currently being verified. The two components of this description are a separation logic assertion describing the current memory and a predicate vfEqv describing the current variable environment. $\mathsf{vfEqv}\ S\ \phi\ v$ holds if the set S is a subset of the domain of the variable environment v of the current procedure and for all Cminor variables x in the domain of the partial function ϕ we have that $\phi(x) = v(x)$.

Fig. 9 shows the rough sequence of steps that $\mathsf{vcSteps}$ carries out when the start of the current statement is a load of a variable. The complex sequence is needed because the right hand side of the statement might be an expression of arbitrary complexity. The two hypothesis V and H above the line describe the

V : vfEqv $\{\mathtt{x}, \mathtt{y}\}$ $\{(\mathtt{x} \rightsquigarrow a)\}$ $(\pi_2\ \sigma)$
H : $(A * a \mapsto v * B)$ $(\pi_1\ \sigma)$

---

stmPre $(\mathtt{y} := [\mathtt{x}]\ ;\ s)\ P\ \sigma$
stmPre $(\mathtt{y} := [\mathtt{x}])$ (stmPre $s\ P$) σ
do $v' \leftarrow$ eval$(\sigma, [\mathtt{x}])$; do $\sigma' \leftarrow$ setVar$(\sigma, \mathtt{y}, v')$; stmPre $s\ P\ \sigma'$
do $v' \leftarrow$ eval$(\sigma, [a])$; do $\sigma' \leftarrow$ setVar$(\sigma, \mathtt{y}, v')$; stmPre $s\ P\ \sigma'$
do $v' \leftarrow$ Some(v); do $\sigma' \leftarrow$ setVar$(\sigma, \mathtt{y}, v')$; stmPre $s\ P\ \sigma'$
do $\sigma' \leftarrow$ setVar$(\sigma, \mathtt{y}, v)$; stmPre $s\ P\ \sigma'$

Fig. 10. Program logic tactics: examining the state

V' : vfEqv $\{\mathtt{x}, \mathtt{y}\}$ $\{(\mathtt{x} \rightsquigarrow a), (\mathtt{y} \rightsquigarrow v)\}$ $(\pi_2\ \sigma')$
H : $(A * a \mapsto v * B)$ $(\pi_1\ \sigma')$

---

stmPre $s\ P\ \sigma'$

Fig. 11. Program logic tactics: updating the state

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, then for an assignment. Next the tactic determines that the value of the variable x is a by examining the hypothesis V. After that is done, the load operation is entirely in terms of values, so the tactic will examine the hypothesis H to determine that address a contains value v. The relevant binding $a \mapsto v$ can occur as any subtree of the separation logic assertion. Now the do-notation can be reduced away.

Now it remains to show that the variable update is safe, and to reflect the result of the update in V, the description of the current variable environment. y is an element of the set that is the second argument of V, so it is safe to write to the variable y. The tactics then define a new state σ' that is the result of updating y to v in σ, then create a description V' of the variable environment of the new state. This results in the new proof state shown in Fig. 11. The memory in σ' is the same as σ, so H is unchanged. If the operation had instead been a store to memory, H would have been updated instead of V. The tactics can now begin the same unfolding process for s.

This may seem like a lengthy series of steps, but it is invisible to the user. The tactics will get "stuck" at various points that require user intervention. For instance, at a loop an invariant must be supplied. They can also get stuck if the program is accessing memory at an address a but the description of memory does not contain an assertion of the form $a \mapsto v$. In this case, the tactics in the previous section can be applied to manipulate the assertion until it does, then vcSteps can be invoked again to pick up where it left off.

6 Implementation

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. **??**) is about 4700 lines. The definition of separation logic assertions and associated lemmas (discussed in Sect. 2) are about 1100 lines, while the tactics are about 3000 lines. Finally, the program logic (discussed in Sect. **??**), which includes its definition, proofs, and associated tactics, is about 2000 lines.

7 Application

We have used the tactics described in this paper to verify the safety and completeness of a Cheney copying garbage collector [7] 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.* [24] who verify a Cheney collector in 2000 lines using a decompilation based approach, and Hawblitzel *et al.* [**?**] who use an automated theorem prover to automatically verify a collector that is realistic enough to be used for real C# benchmarks.

8 Related Work

Appel's unpublished note [8] describes tactics implemented in the Coq proof assistant for manual verification in a proof assistant using separation logic. These tactics do not support as many manipulations of separation logic assertions, but they do support matching and extraction of existentials and pure assertions $!P$. There is also an application tactic that is similar to our rewriting tactic, but it 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.

The tactics also step procedures one statement at a time. While Appel also applies a "two-level approach" that attempts to pull things out of separation logic assertions to leverage existing tactic infrastructure, our approach is more aggressive about this, lifting out expression evaluation. This allows our approach to avoid reasoning about whether expressions in assertions are pure or not.

We can give a rough comparison of proof sizes, using the standard in-place list reversal benchmark. Ignoring comments and the definition of the program, by the count of `wc` Appel uses 200 lines and 795 words to verify this program [8]. With our tactics, ignoring comments, blank lines and the definition of the program, our verification takes 68 lines and less than 400 words.

Appel and Blazy [16] discuss a similar separation logic for Cminor which is structured in terms of conventional Hoare logic "triples" instead of as a precondition generator, and requires more restrictions about the purity of expressions. Power and Webster [17] describe a deep embedding of linear logic in Coq along with a couple of very primitive tactics. Affeldt and Marti [14] 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.* [18] 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 specialized tactics for separation logic connectives.

Other work has focused on automated verification of lightweight separation logic specifications. Smallfoot [9] is an automated tool for verifying lightweight separation logic specifications of programs. This approach has been used as the basis for certified separation logic decisions procedures in Coq [19] and HOL [20]. Calcagno *et al.* [10] 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 [13, 21, 22] using either deep or shallow embeddings. Expressing assertions via more conventional propositions enables the use of powerful preexisting tactics. Another approach to program verification decompiles imperative programs into functional programs that are more amenable to analysis in a proof assistant [23, 24].

9 Future Work and Conclusion

The tactics we have described in this paper provide a solid foundation for the use of separation logic in a proof assistant but further automation could further reduce the size of proof scripts. Integrating a Smallfoot-like decision procedure into our tactics would automate reasoning about standard data structures. Improved reasoning about modular arithmetic would also be useful.

Our tactics could also be adopted to other similar logics, such as linear logic. This will require proving new commutative and associative lemmas for these logics, but the top level tactics are fairly independent of these low level details. Most of the work would be remove special cases that only apply to separation logic, such as matching rules for the contains predicate $\mapsto$.

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 powreful enough to verify a garbage collector.

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

References

1. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS '02, Washington, DC, USA, IEEE Computer Society (2002) 55–74
2. Girard, J.Y.: Linear logic. Theoretical Computer Science **50** (1987) 1–102
3. Birkedal, L., Torp-Smith, N., Reynolds, J.C.: Local reasoning about a copying garbage collector. In: POPL '05, New York, NY, USA, ACM Press (2004) 220–231
4. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: PLDI '07, New York, NY, USA, ACM (2007) 468–479
5. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr
6. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL '06, ACM Press (2006) 42–54
7. Cheney, C.J.: A nonrecursive list compacting algorithm. Communications of the ACM **13**(11) (1970) 677–678
8. Appel, A.W.: Tactics for separation logic. http://www.cs.princeton.edu/ appel/papers/septacs.pdf (January 2006)
9. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: 4th Formal Methods for Components and Objects. (2005) 115–137
10. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL '09, New York, NY, USA, ACM (2009) 289–300
11. McCreight, A.: The Mechanized Verification of Garbage Collector Implementations. PhD thesis, Yale University, New Haven, CT, USA (2008)
12. Paulin-Mohring, C.: Inductive definitions in the system Coq—rules and properties. In: Proc. TLCA. Volume 664 of LNCS. (1993)
13. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In Slind, K., Bunker, A., Gopalakrishnan, G., eds.: TPHOLs '04. Volume 3223 of LNCS., Springer (2004) 305–320
14. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In: ICFEM 2006: Eighth International Conference on Formal Engineering Methods. (2006)
15. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: TACS '97. Springer-Verlag LNCS 1281, Springer-Verlag (1997) 515–529
16. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: TPHOLs '07. Volume 4732 of LNCS., Springer (2007) 5–21
17. Power, J., Webster, C.: Working with linear logic in Coq. In: TPHOLs '99 (work-in-progress paper). (September 1999)
18. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: POPL '07, New York, NY, USA, ACM (2007) 97–108

19. Marti, N., Affeldt, R.: A certified verifier for a fragment of separation logic. In: 9th JSSST Workshop on Programming and Programming Languages (PPL 2007). (2007)
20. Tuerk, T.: A separation logic framework in HOL. In Otmane Ait Mohamed, C.M., Tahar, S., eds.: TPHOLs '08: Emerging Trends Proceedings. (08 2008) 116–122
21. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1-2) (2005) 200–227
22. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOLs '08, Berlin, Heidelberg, Springer-Verlag (2008) 134–149
23. Filliâtre, J.C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. Computer Aided Verification (2007) 173–177
24. 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)