Tactics for Separation Logic

EARLY DRAFT

Andrew W. Appel
INRIA Rocquencourt & Princeton University

January 13, 2006

Abstract

Separation logic is a Hoare logic for programs that alter pointer data structures. One can do machine-checked separation-logic proofs of interesting programs by a semantic embedding of separation logic in a higher-order logic such as Coq or Isabelle/HOL. However, since separation is a linear logic—actually, a mixture of linear and nonlinear logic—the usual methods that Coq or Isabelle use to manipulate hypotheses don't work well. On the other hand, one does not want to duplicate in linear logic the entire libraries of lemmas and tactics that are an important strength of the Coq and Isabelle systems. Here I demonstrate a set of tactics for moving cleanly between classical natural deduction and linear implication.

1 Introduction

In proving correctness properties of imperative programs, Hoare logic is useful. And if these programs manipulate (allocate, free, initialize update) pointers into the heap, *separation logic* is expressive and convenient. Separation logic [4, 5] is a form of Hoare logic whose operators reason about the domains of memory regions, and in particular the disjointness of those domains; this is useful for proving that a store through one pointer will not affect a load through another pointer.

In this logic, a *separating conjunction* $P * Q$ holds on a heap when the assertion P holds on one portion of the heap, Q holds on another portion, these two portions are disjoint, and their union forms the entire heap reasoned about by $P * Q$. This is therefore a linear logic, in that in general $P * Q$ does not entail $P * P * Q$ or vice versa.

When doing machine-checked proofs of imperative programs, one faces a choice: one could implement Hoare logic (or separation logic) directly (or directly in a logical framework such as Twelf or Isabelle); or one could define the operators of the Hoare or separation logic inside a higher-order logic (such as Coq or Isabelle/HOL). Although the former approach appears to enjoy a nice purity and simplicity, I will advocate the two-level approach. Even when proving an imperative program, much of the reasoning is not about memory cells but concerns the abstract mathematical objects that the program's data structures represent. Lemmas about those objects are most conveniently proved in a general-purpose higher-order logic, especially when there are large and well-documented libraries of lemmas and tactics.

In this two-level approach, the "upper level" is Hoare (or separation) logic, and the "lower level" is higher-order logic (or the calculus of constructions, etc.; from now on I will use HOL to indicate either of these logics). Affeldt and Marti have defined just such an embedding [2] and have used it to prove the correctness of a memory manager [3].

In the two-level approach one needs to define lemmas and tactics to conveniently move between the levels. I will show that, in a tactical prover such as Coq or Isabelle/HOL, a simple style of proof that works well for Hoare logic does not work well for separation logic, and that a new approach is desirable.

2 Embedding Hoare logic in HOL

The cleverest, most beautiful, and ultimately the most misleading aspect of Hoare's notation is his *pun* which confuses program expressions with logical formulas. In the triple $\{P\}C\{Q\}$ with precondition P, command C, and postcondition Q, the same boolean expressions and integer variables can appear in P or Q as a logical formula and in C as a part of the program. For example,

$$\{a = b \cdot 2 + 1\}\ b \leftarrow b \cdot 2\ \{a = b + 1\}$$

the expression $b \cdot 2$ is used both ways. When giving a semantics of Hoare logic, one must undo the confusion: it is more straightforward to say that there is a set of *program variables*, a *state* maps from program variables to *values*, a command C relates a *before* state to an *after* state.

I'll assume here a simple presentation where values are just integers and program variables are also represented as integers. The language of commands has a syntax including integer expressions *Expr*, boolean expressions *Exprb*, and commands *Cmd* with the usual operators. Then we define, by syntactic induction, evaluation functions on program fragments:

$$\begin{array}{lcl} \mathrm{eval} & : & \mathit{Expr} \rightarrow \mathit{State} \rightarrow \mathit{Value} \\ \mathrm{evalb} & : & \mathit{Exprb} \rightarrow \mathit{State} \rightarrow \mathit{Bool} \\ \mathrm{exec} & : & \mathit{State} \rightarrow \mathit{Cmd} \rightarrow \mathit{State} \rightarrow \mathit{Bool} \end{array}$$

In this paper I'll confuse the boolean values with the logical propositions to simplify the presentation; all the underlying results are implemented more formally in Coq.

Are the assertions such as P and Q simply boolean expressions from the programming language? If so, we can define the Hoare triple

$$\begin{array}{l} \{P\}C\{Q\} \equiv \\ \forall s : \mathit{State}.\ \ \mathrm{evalb}\, P\, s\ \Rightarrow \exists s'.\ \mathrm{exec}\, s\, C\, s'\ \wedge\ \mathrm{evalb}\, Q\, s' \end{array}$$

This is a naive definition: it doesn't account for the possible nondeterminism or nontermination of the command C; but what I will say in this paper is largely independent of the exact form of the axiomatic semantics.

The real problem with assertions-as-boolean-expressions is that, to prove properties of nontrivial programs, the assertion language must be fairly powerful; it must include quantifiers, which don't usually appear in the boolean-expression language of programming languages.

While we could augment the syntax of the programming language, it is more straightforward to accept the fact that the logic for reasoning about a program should be more expressive than the programming language. Then we should consider assertions not as boolean expressions but as predicates on states, that is, functions from states to truth values. Now our Hoare triple is,

$$\{P\}C\{Q\} \equiv \forall s : \mathit{State}.\ \ P\, s\ \Rightarrow \exists s'.\ \mathrm{exec}\, s\, C\, s' \wedge Q\, s'$$

But now when we write the triple for $b \leftarrow b \cdot 2$ we have,

$$\{\lambda s.\, sa =\ 2(sb) + 1\} b \leftarrow b \cdot 2 \{\lambda s.\, sa = sb + 1\}$$

or perhaps

$$\{\mathrm{evalb}(a = b \cdot 2 + 1)\} b \leftarrow b \cdot 2 \{\mathrm{evalb}(a = b + 1)\}$$

In the first of these formulations we have lost the identification of logical formulas with program expressions; in the second we find that we cannot easily quantify values such as a, which are buried inside a syntactic expression, not a logical formula.

Either way, with assertions-as-predicates, we have the expressive power to write quantified formulas; we can use the values of variables and we can evaluate programming-language expressions:

$$\begin{array}{l} \{\lambda s.\, \exists y.\, y = 2(sa) \wedge sb = y + 1\} \\ b \leftarrow b \cdot 2 \\ \{\lambda s. \exists z. \mathrm{evalb}(b = a + 1) s\} \end{array}$$

This seems clumsier than Hoare's pun; but what happens when we embed it in a higher-order-logic tactical prover?

singleton	$e_1 \mapsto e_2$	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\mathrm{eval}\, e_1\, s\} \wedge h(\mathrm{eval}\, e_1\, s) = \mathrm{eval}\, e_2\, s$
empty	$\mathbf{emp}$	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\,\}$
conjunction	$P * P$	$\equiv$	$\lambda sh.\exists h_1 h_2.h = h_1 \cup h_2 \wedge \mathrm{dom}\, h_1 \cap \mathrm{dom}\, h_2 = \{\,\} \wedge Psh_1 \wedge Qsh_2$

Figure 1: Primitives of separation logic

3 Tactical manipulation of Hoare logic

In a tactical prover such as Coq or Isabelle/HOL, one manipulates proof goals of the form,

$$\frac{\begin{array}{l} x_1 : t_1 \\ \vdots \\ x_m : t_m \\ H_1 : e_1 \\ \vdots \\ H_n : e_n \end{array}}{e}$$

where the x_i are assumed variables, t_i are the types of these variables, e_i are hypotheses, H_i are the names of the hypotheses, and e is the conclusion.

When manipulating Hoare-logic assertions, a typical situation (arising, for example, from the *while* rule or the strenghtening-precondition rule) is that we need to prove $\forall s.Ps \Rightarrow Qs$. Suppose P is the conjunction of several terms P_i, and Q is the conjuction of Q_j. It's a simple matter to make lemmas and tactics that break up the goal $\forall s.Ps \Rightarrow Qs$ into subgoals such as

$$\frac{\begin{array}{l} s : \mathit{State} \\ H_1 : P_1 \\ \quad\vdots \\ H_n : P_n \end{array}}{Q_1}$$

Such subgoals can be proved using the normal lemmas and tactics in the theorem-prover's library. That is, the embedding of ordinary Hoare logic in HOL does not necessarily require specialized tactics.

4 Embedding separation logic

Separation logic [4, 5] is a form of Hoare logic that treats *local variables* differently from *memory.* It can be modeled by dividing the *state* into two parts, the *store* (a mapping from *local variable names* to *values*) and the *heap* (a mapping from *locations* to *values*).

We will follow Affeldt and Marti's embedding of separation logic in Coq [2], which in turn follows Reynolds's presentation [5]. We will use s to range over stores and h to range over heaps. But since the heap is addressable by pointers with address arithmetic, we might as well just admit that locations are just integer values, i.e. *locations=values.*

We assume the programming language has *expressions*[1] that involve only the values of local variables (i.e., the store) but not the heap; and *commands* that can fetch/store heap locations to/from local variables. Thus $\mathrm{eval}\, e\, s$ evaluates an expression (in a store) to an integer, but $\mathrm{exec}\,(s_1, h_1)\, c\,(s_2, h_2)$ is the execution of a command relating an old state to a new state.

For separation logic we emphasize that each heap is a *finite* mapping with a particular domain. We say two heaps are disjoint if their domains are disjoint, and we can form the union of two (disjoint) heaps to form a new heap.

Each assertion of separation logic is a predicate on a store and a heap. The primitives are shown in Figure 1.

In addition to assertions about the heap, we can make arbitrary *pure* assertions about the store, such

[1] A semiformal mathematical presentation of expression syntax would have productions such as $e{::=}v|n|e{+}e$ etc., while a formal theorem-prover embedding would have explicit coercions $e{::=}\mathrm{var\_e}\ v\ |\mathrm{int\_e}\ \ n\ |\ e_1\ \ +_{\mathrm{e}}\ \ e_2$. In this paper I will use the semiformal style; that is, I will leave out coercions var_e and int_e that actually appear in the machine-checked proofs.

as $\lambda sh.\exists y.\, y = 2(sa) \wedge sb = y+1$. We can use ordinary conjuction $A \wedge B$, to combine pure assumptions (or pure with impure), but impure assuptions should be combined only with separating conjunction $P * Q$.

What happens when we apply a natural-deduction tactical prover to this semantic embedding of separation logic? Let us take a goal such as

$$\forall sh.\ \lambda sh.As \wedge (P * Q * R)sh \ \Rightarrow (Bs \wedge Ush) * Vsh$$

and apply routine tactics to expand the definitions and introduce the quantified variables:

$$\begin{array}{l}
s : \mathit{Store} \qquad h : \mathit{Heap} \\
H_1 : As \\
h_p : \mathit{Heap} \quad h_q : \mathit{Heap} \quad h_r : \mathit{Heap} \quad h_{pq} : \mathit{Heap} \\
H_{pq} : h_{pq} = h_p \cup h_q \\
H_h : h = h_{pq} \cup h_r \\
H_{pq'} : h_p \cap h_q = \{\,\} \\
H_{h'} : h_{pq} \cap h_r = \{\,\} \\
H_2 : Psh_p \quad H_3 : Qsh_q \quad H_4 : Rsh_r \\
\hline
Bs \wedge \\
\exists h_u h_v.\ h = h_u \cup h_v \wedge h_u \cap h_v = \{\,\} \wedge Ush_u \wedge Vsh_v
\end{array}$$

It's easy to separate this into two subgoals Bs and $\exists h_u h_v.\ h = h_u \cup h_v \wedge h_u \cap h_v = \{\,\} \wedge Ush_u \wedge Vsh_v$, but it is not easy to automatically break up the existential conjunction. In contrast to ordinary Hoare logic with nonlinear conjuction, the nonlinear conjunction of separation logic is not well suited to the assumptions of tactical provers for higher-order logics: that the hypotheses of each goal can be broken into separate named assumptions, and the conclusion can be split to separate subgoals. The proliferation of hypotheses $h_p, h_q, h_r, h_{pq}, H_{pq}, H_h, H_{pq'}, H_{h'}$ makes this approach unattractive.

In fact, the whole purpose of separation logic is to encapsulate and hide propositions about disjointness of heap fragments. Any proof in "separation logic" that explicitly manipulates such hypotheses manifests in some sense a failure of the abstraction.

5 An assertion language for separation logic

Comfortable theorem-proving in separation logic should have these characteristics: (1) Hoare-triple reasoning should proceed naturally, as Reynolds does in his semiformal proofs, and should avoid explicit reasoning about heaps except through the separating conjuction operator. (2) Purely mathematical reasoning should proceed naturally, as it does in ordinary proofs in higher-order logic, and take advantage of existing libraries of lemmas and tactics. (3) There should be natural transitions between the two levels.

In Figure 2 I introduce some operators to support such a style of proof. Given a boolean expression e of the programming language, the assertion $!e$ represents that e evaluates to true on the store *and the heap is empty.* Given a formula e of logic (independent of the store or heap), $!!e$ represents that e is true *and the heap is empty.* Finally, given a separation-logic assertion P, existential $\exists x : \tau.P$ indicates that, given a store and heap, there exists an x such that Px holds on that store and heap. The type τ may be any type that the underlying logic can ordinarily quantify over.

These operators come equipped with certain tactics and lemmas, which I will explain in the course of two two running examples: first, a program that swaps the contents of two memory locations in the heap; second, the obligatory (for separation logic papers) in-place list-reversal algorithm.

Each tactic is supported by a collection of lemmas that are proved with respect to the Affeldt-Marti specification of separation logic; that is, the tactics are sound.

5.1 Swap

1 $\forall uvijxy,$
2 $\text{var.set}[u, v] \ \Rightarrow$
3 $\{i \mapsto x \ * \ j \mapsto y\}$
4 $u \leftarrow [i];\ v \leftarrow [j];\ [i] \leftarrow v;\ [j] \leftarrow u$
5 $\{i \mapsto y \ * \ j \mapsto x\}$

Line 1 quantifies over program variables and heap locations. Line 2 asserts that u and v are different variables. Line 3 is the precondition: that the heap contains exactly two locations i, j containing values x, y respectively. Line 4 is the program, where $u \leftarrow [i]$ is a load instruction from location i, and $[i] \leftarrow v$ is a store instruction. Line 5 is the postcondition.

singleton	$e_1 \mapsto e_2$	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\mathrm{eval}\, e_1\, s\} \wedge h(\mathrm{eval}\, e_1\, s) = \mathrm{eval}\, e_2\, s$
empty	**emp**	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\,\}$
conjunction	$P * P$	$\equiv$	$\lambda sh.\exists h_1 h_2. h = h_1 \cup h_2 \wedge \mathrm{dom}\, h_1 \cap \mathrm{dom}\, h_2 = \{\,\} \wedge P s h_1 \wedge Q s h_2$
eval	$!e$	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\,\} \wedge \mathrm{evalb}\, e\, s$
prop	$!!e$	$\equiv$	$\lambda sh.\mathrm{dom}\, h = \{\,\} \wedge e$
exists	$\exists x : \tau.\ P$	$\equiv$	$\lambda sh.\exists x : \tau.\ P s h$

Figure 2: New primitives of separation logic. (Primitives above the line are unchanged.)

The proof takes exactly 8 lines:

```
1 intros.
2 Forward.
3 Forward.
4 Forward.
5 Forward.
6 assert_subst (var_e u == int_e x).
7 assert_rewrite (var_e v == int_e y)
            (fun z => int_e i |-> z).
8 sep_trivial.
```

Line 1 uses the usual Coq `intros` tactic to introduce the variables $uvijxy$ and hypothesis var.set.

Line 2 applies the (new) `Forward` tactic to move forward through atomic statement (i.e., load, store, or heap-independent assignment). Forward is applicable in these conditions:

- to $\{P\}v \leftarrow e; C\{Q\}$ when v is not free in P or e; the remaining proof obligation (subgoal) is $\{P *!(v = e)\}C\{Q\}$.
- to $\{P_1 * (e_1 \mapsto e_2) * P_2\}v \leftarrow [e_1]; C\{Q\}$ when v is not free in e_1, e_2, P_1, P_2; the subgoal is $\{P_1 * (e_1 \mapsto e_2) *!(v = e_2) * P_2\}C\{Q\}$
- to $\{P_1 * (e_1 \mapsto e') * P_2\}[e_1] \leftarrow e_2; C\{Q\}$; the subgoal is $\{P_1 * (e_1 \mapsto e_2) * P_2\}C\{Q\}$.

In all cases, if $;C$ is not present, then the subgoal is of the form $\{P'\}\mathbf{skip}\{Q\}$ or equivalently $P' \Longrightarrow Q$. In all cases, $P_1 * P * P_2$ is shorthand for any tree of separating conjuctions containing the conjunct P.

Thus, after line 2 the proof obligation is

$$\begin{array}{l} u, v : \mathit{Variable} \qquad i, j, x, y : \mathit{Integer} \\ H : \mathrm{var.set}[u, v] \\ \hline \{i \mapsto x\ *\ !(u = x)\ *\ j \mapsto y\} \\ v \leftarrow [j];\ [i] \leftarrow v;\ [j] \leftarrow u \\ \{i \mapsto y\ *\ j \mapsto x\} \end{array}$$

After another Forward (line 3) we have

$$\begin{array}{l} \{i \mapsto x\ *\ !(u = x)\ *\ j \mapsto y\ *\ !(v = y)\} \\ [i] \leftarrow v;\ [j] \leftarrow u \\ \{i \mapsto y\ *\ j \mapsto x\} \end{array}$$

and after two more, we have

$$i \mapsto v *!(u = x) * j \mapsto u *!(v = y) \implies i \mapsto y * j \mapsto x$$

Line 6 applies the (new) `assert_subst` tactic, named by analogy with Coq's `subst` tactic. Given a hypothesis $H : v = e$, `subst` will replace all occurrences of (logic) variable v with the (logic) expression e, and will delete H. Similarly, given any one of the following goals,

$$\begin{array}{l} \{P_1 *!(e_1 = e_2) * P_2\}C\{Q\} \\ \{P_1 *!(e_2 = e_1) * P_2\}C\{Q\} \\ P_1 *!(e_1 = e_2) * P_2 \implies Q \\ P_1 *!(e_2 = e_1) * P_2 \implies Q \end{array}$$

the application of `assert_subst`$(e_1 = e_2)$ will produce a goal where all occurrences of e_1 in P_1, P_2 are replaced by e_2, and the equation $!(e_1 = e_2)$ is deleted. No change is made to Q.

After line 6 we have,

$$i \mapsto v\ *\ j \mapsto x\ *\ !(v = y) \implies i \mapsto y\ *\ j \mapsto x$$

Now `assert_subst` could be used again, but I will illustrate `assert_rewrite` instead. Like the Coq `rewrite` tactic, `assert_rewrite` makes one replacement rather than every possible replacement. The first argument is the (programming-language) equality to be used as the rewrite rule; the second describes a context in which to perform the rewrite. Like `rewrite`, it does not remove the equation from the hypotheses. The result in this case is,

$$i \mapsto y \ * \ j \mapsto x \ * \ !(v = y) \implies i \mapsto y \ * \ j \mapsto x$$

Line 8 applies the `sep_trivial` tactic. The goal follows trivially from dropping the pure conjunct $!(v = y)$ from the left-hand-side. Unlike impure assertions containing $e \mapsto e'$ which cannot just be dropped or duplicated, any assertion of the form $!e$ or $!!e$ is (formally) about the empty heap, so we have the lemma $P*!e \implies P$. Sep_trivial is able to accommodate any rearrangement of atomic impure assertions, any dropping or duplication of impure tactics, and the construction (on the right) of trivial pure assertions such as $!!(x = x)$.

5.2 In-place list reverse

We can choose to represent a list cell (h, t) in separation logic as $x \mapsto h \ * \ x + 1 \mapsto t$ at address $x \neq 0$, and we can use 0 to represent the empty list. Given a list whose root is in variable v, the following program reverses all the tail-pointers in place, leaving a pointer to the reversed list in variable w:

$w \leftarrow 0;$
while $v \neq 0$
  do $(t \leftarrow [v + 1]; \ [v + 1] \leftarrow w; \ w \leftarrow v; \ v \leftarrow t)$

To describe the precondition and postcondition, we make an inductive assertion (contents l x) meaning that the sequence of integers l : list(*Integer*) is represented in memory as a list with root address x : *Integer*.

contents_nil : $\forall l, x.$
$!!(l = 0) \ * \ !!(x = 0) \implies \text{contents } l \ x$

contents_cons : $\forall l, x.$
$\exists h. \ \exists t. \ \exists p.$
$!!(l = h{::}t) \ * \ !!(x \neq 0) \ * \ (x \mapsto h) \ * \ (x + 1 \mapsto p)$
$\ * \ \text{contents } t \ p$
$\implies \text{contents } l \ x$

Now we can state the precondition and postcondition of the program:

var.set$[w, v, t] \ \Rightarrow$
$\{\text{contents } l \ v\}$
$w \leftarrow 0;$
while $v \neq 0$
  do $(t \leftarrow [v + 1]; \ [v + 1] \leftarrow w; \ w \leftarrow v; \ v \leftarrow t)$
$\{\text{contents } (\text{rev } l) \ w\}$

We will make use of the loop invariant,

$$Inv =$$
$$\exists l_1 \exists l_2. \text{contents } l_2 \ v \ * \ \text{contents } (\text{rev } l_1) \ w * !!(l = l_1 + l_2)$$

where $l_1 + l_2$ is list concatentation.

The proof relies on some auxiliary lemmas:

Lemma sep_list_0_nil.

$$\text{contents } l \ 0 \implies !!(l = 0)$$

Lemma inde_contents. If x is an integer constant, then the contents $l \ x$ has no free variables (evaluates the same in any store).

Lemma begin_while.

$$\text{contents } l \ v \ * \ !(w = 0) \implies Inv$$

Lemma end_while.

$$Inv \ * \ !\neg(v \neq 0) \implies \text{contents } (\text{rev } l) \ w$$

Lemma list_cons_lemma.

$$w \mapsto h \ * \ (w + 1) \mapsto n \ * \ !(w \neq 0) \ * \ \text{contents } l \ n$$
$$\implies \text{contents } (h{::}l) \ w$$

Lemma list_fetchable.

$$\text{contents } l \ e \ * \ !(e \neq 0) \implies$$
$$\exists h \ \exists t \ \exists p$$
$$!(l = h{::}t) \ * \ \text{contents } t \ p \ *$$
$$e \mapsto h \ * \ (e + 1) \mapsto p$$

None of these lemmas is particularly novel; similar lemmas could be proved (and have been proved [1]) in another separation-logic-in-HOL system. What is new is that they can be proved smoothly, without directly manipulating heap-disjointness hypotheses; also that they can be written without λsh.

We will show more of the new tactics in the proof of the main theorem. After applying `Forward` we have the proof obligation,

$$
\begin{array}{l}
\{\text{contents } l\ v\ *\ !(w=0)\}\\
\textbf{while } v \neq 0\\
\quad \textbf{do } (t \leftarrow [v+1];\ [v+1] \leftarrow w;\ w \leftarrow v;\ v \leftarrow t)\\
\{\text{contents } (\text{rev}\, l)\ w\}
\end{array}
$$

The traditional while-loop axiom is inconvenient for our tactics to manipulate, because it uses a mixture of ordinary conjuction $\wedge$ with separating conjunction $*$. The purpose of our assertion-forms $!e$ and $!!e$ is to make assertions that can be meaningfully combined using $*$. Thus we apply a (new) while-loop lemma,

$$
\frac{\begin{array}{c} P \implies I \\ \{I*!B\}C\{I\} \\ I*!\neg B \implies Q \end{array}}{\{P\}\textbf{while } B \textbf{ do } C\{Q\}}
$$

using our loop-invariant *Inv*, and we obtain three subgoals. The first and third are exactly the lemmas begin_while and end_while; the remaining subgoal is,

$$
\begin{array}{l}
\{(\exists l_1 \exists l_2.\text{contents}\, l_2\, v\ *\ \text{contents}\,(\text{rev}\, l_1)\, w\ *\\
!!(l = l_1 + l_2))\ *\ !(v \neq 0)\}\\
t \leftarrow [v+1];\ [v+1] \leftarrow w;\ w \leftarrow v;\ v \leftarrow t\\
\{\mathit{Inv}\}
\end{array}
$$

We have a lemma,

$$
\frac{\exists x : \tau.\ \{P\}C\{Q\}}{\{\exists x : \tau.P\}C\{Q\}}
$$

which is almost applicable here, except that the $\exists l_1$ in our current precondition is on the left side of a conjunction. No matter: the `Exists_left` tactic pulls the leftmost existential out of any separating-conjunctions and out of the precondition (or out of the left-hand side of a $\implies$ entailment):
`Exists_left l_1; Exists_left l_2.`

Now within the precondition we have the assertion $!!(l = l_1 + l_2)$; and the lemma,

$$
\frac{e\ \Rightarrow\ \{P\}C\{Q\}}{\{!!e\ *\ P\}C\{Q\}}
$$

The tactic `extract_prop` uses this lemma with associative-commutative laws to extract the leftmost pure proposition from the precondition (or l.h.s of $\implies$); we apply it here to leave the goal,

$$
\begin{array}{l}
\{(\text{contents}\, l_2\, v\ *\ \text{contents}\,(\text{rev}\, l_1)\, w * !(v \neq 0)\}\\
t \leftarrow [v+1];\ [v+1] \leftarrow w;\ w \leftarrow v;\ v \leftarrow t\\
\{\mathit{Inv}\}
\end{array}
$$

At this point two of the left-hand conjuncts match the left-hand-side of the list_fetchable lemma. We should be able to use a rule of the form $(P \implies Q) \Rightarrow (Q * R \implies S) \Rightarrow (P * R \implies S)$, along with appropriate associate-commutative rearrangements of $*$, to apply this lemma; that's precisely what the next tactic does:
`assert_apply (list_fetchable l2 v).`

The resulting subgoal has three existentials and one proposition on the left-hand side; we use `Exists_left` and `extract_prop` to obtain the goal,

$$
\frac{H_0 : l = l_1 + l_2 \qquad H_1 : l_2 = h{::}t_0}{\begin{array}{l}
\{\text{contents}\, t_0\, p\ *\ v \mapsto h\ *\ (v+1) \mapsto p\ *\\
\ \text{contents}\,(\text{rev}\, l_1)\, w\ *\ !(v \neq 0)\}\\
t \leftarrow [v+1];\ [v+1] \leftarrow w;\ w \leftarrow v;\ v \leftarrow t\\
\{\mathit{Inv}\}
\end{array}}
$$

Now the left-hand side has the conjuncts necessary to move `Forward` twice, leaving

$$
\frac{H_0 : l = l_1 + l_2 \qquad H_1 : l_2 = h{::}t_0}{\begin{array}{l}
\{\text{contents}\, t_0\, p\ *\ v \mapsto h\ *\ (v+1) \mapsto w\ *\\
\ !(t = p)\ *\ \text{contents}\,(\text{rev}\, l_1)\, w\ *\ !(v \neq 0)\}\\
w \leftarrow v;\ v \leftarrow t\\
\{\mathit{Inv}\}
\end{array}}
$$

We cannot move `Forward` again, because that tactic requires that the l.h.s. variable of the assignment (w) must not appear free in the precondition. We would like to replace w in the precondition by its integer value; first we use a lemma that any expression has a value in any store:

$$\frac{\forall n.\ \{!(e == n) * P\}C\{Q\}}{\{P\}C\{Q\}}$$

The tactic looks like this:
`apply(expr_has_value w); intro n.`
and now we have the conjunct $!(w = n)$ in the precondition. We apply
`assert_subst(w=n)` to obtain,

$$\frac{H_0 : l = l_1 + l_2 \qquad H_1 : l_2 = h{::}t_0}{\begin{array}{l}\{\text{contents}\, t_0\, p\ *\ v \mapsto h\ *\ (v+1) \mapsto n\ * \\ \ !(t = p)\ *\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(v \neq 0)\} \\ w \leftarrow v;\ v \leftarrow t \\ \{\mathit{Inv}\}\end{array}}$$

and since w is no longer free in the precondition, we can move `Forward.`

$$\frac{H_0 : l = l_1 + l_2 \qquad H_1 : l_2 = h{::}t_0}{\begin{array}{l}\{\text{contents}\, t_0\, p\ *\ v \mapsto h\ *\ (v+1) \mapsto n\ * \\ \ !(t = p) * \text{contents}\,(\text{rev}\, l_1)\, n * !(v \neq 0) * !(w = v)\} \\ v \leftarrow t \\ \{\mathit{Inv}\}\end{array}}$$

Now we can apply
`assert_subst(v=w); Forward.`
to obtain

$$\begin{array}{c}\text{contents}\, t_0\, p\ *\ w \mapsto h\ *\ (w+1) \mapsto n\ *\ !(t = p)\ * \\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(w \neq 0)\ *\ !(v = t) \\ \Longrightarrow\ \mathit{Inv}\end{array}$$

Using `assert_subst` we substitute v for t and then v for p, then unfold Inv to get,

$$\frac{H_0 : l = l_1 + l_2 \qquad H_1 : l_2 = h{::}t_0}{\begin{array}{l}\text{contents}\, t_0\, v\ *\ w \mapsto h\ *\ (w+1) \mapsto n\ * \\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(w \neq 0) \\ \ \Longrightarrow \\ \exists l_3 \exists l_4 . \text{contents}\, l_4\, v\ *\ \text{contents}\,(\text{rev}\, l_3)\, w\ * \\ \quad !!(l = l_3 + l_4)\end{array}}$$

By analogy with the Coq `exists` tactic, we can use `Exists_right` to instantiate an existential in the r.h.s. of a sequent; we use it twice:

```
Exists_right (l1+(h::nil)).
Exists_right t0.
```

Now we can use the standard Coq `subst l2` to apply the hypothesis H_1; and we can `replace (rev(h::l1)) with (rev l1 + (h::nil))` which Coq can verify using its `tauto` tactic. This leaves,

$$\frac{H_0 : l = l_1 + (h{::}t_0)}{\begin{array}{l}\text{contents}\, t_0\, v\ *\ w \mapsto h\ *\ (w+1) \mapsto n\ * \\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(w \neq 0) \\ \ \Longrightarrow \\ \text{contents}\, t_0\, v\ *\ \text{contents}\,(\text{rev}\,(l_1 + (h{::}\text{nil})))\, w\ * \\ !!(l = l_1 + (h{::}\text{nil}) + t_0)\end{array}}$$

Using standard Coq lemmas and tactics, $l_1 + (h{::}\text{nil}) + t_0$ can be rewritten in two lines to $l_1 + (h{::}t0)$.

$$\frac{H_0 : l = l_1 + (h{::}t_0)}{\begin{array}{l}\text{contents}\, t_0\, v\ *\ w \mapsto h\ *\ (w+1) \mapsto n\ * \\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(w \neq 0) \\ \ \Longrightarrow \\ \text{contents}\, t_0\, v\ *\ \text{contents}\,(h{::}(\text{rev}\, l_1))\, w\ * \\ !!(l = l_1 + (h{::}t_0))\end{array}}$$

At this point, the following two lines would finish the proof:

```
assert_apply(list_cons_lemma n (rev l1) w h).
sep_trivial.
```

but instead I will illustrate a different approach. The tactic `sep_trivial` can perform any associative-commutative rearrangement of the impure terms, plus any trivial deletion, insertion, or duplication of the pure terms. In our current situation, only some parts of the goal are trivial; that is, $!!(l = l_1 + (h{::}t_0))$ appears as a hypothesis H_0 and on the right; and $\text{contents}\, t_0\, v$ appears on both sides. If we apply `sep_trivial` now, all the trivial parts are removed, leaving just,

$$\frac{H_0 : l = l_1 + (h{::}t_0)}{\begin{array}{l}w \mapsto h\ *\ (w+1) \mapsto n\ * \\ \text{contents}\,(\text{rev}\, l_1)\, n\ *\ !(w \neq 0) \\ \ \Longrightarrow \\ \text{contents}\,(h{::}(\text{rev}\, l_1))\, w\end{array}}$$

Once again, `assert_apply` followed by `sep_trivial` will finish the proof. In this case, the early application of `sep_trivial` serves merely to make the proof goal more readable.

5.3 Induction

The contents predicate is inductive; in Coq it is written as,

```
Inductive contents: list Z -> Z -> assert :=
 | contents_nil: forall l x,
    !!(l = nil) ** !! (x = 0)
    ==> contents l x
 | contents_cons:  forall l  x,
    (Exists h, Exists t, Exists ptr,
      !! (l = h::t) **
      (int_e x +e int_e data |-> int_e h) **
      (int_e x +e int_e next |-> int_e ptr) **
      !! (x <> 0%Z)
      ** contents t  ptr)
     ==> contents l x.
```

With inductive predicates in Coq, one normally uses tactics such as `induction` and `inversion`. It is useful to define a new tactic `sep_inversion`, as I will show.

When we define a new predicate such as contents it is usually necessary to prove a lemma about its free variables, so that we can apply `Forward` when contents appears in a precondition. In this case, we prove that for any integer literal n and any set of variables $vars$, the assertion $\text{contents}\,l\,n$ is independent of $vars$.

Such proofs make use of a lemma `inde_intro`,

$$\frac{\forall x n, x \in vars \;\Rightarrow\; \{P\}x \leftarrow n\{P\}}{\text{inde } vars\ P}$$

Lemma inde_contents:
$\forall vars\, l\, n.\ \text{inde } vars(\text{contents}\, l\, n)$.

The proof starts by the standard Coq `induction` tactic followed by various introduction tactics: `induction l; intros; apply inde_intro; intros v i H`. This leaves two subgoals,

$$\frac{H : v \in vars}{\{\text{contents nil } n\}\ v \leftarrow i\ \{\text{contents nil } n\}}$$

$$\frac{H : v \in vars}{\{\text{contents } (a{::}l)\ n\}\ v \leftarrow i\ \{\text{contents } (a{::}l)\ n\}}$$

Consider the first case. We can strengthen the precondition to $!!(\text{nil} = \text{nil}) * !!(n = 0)$, leaving as one of the subgoals, $\text{contents nil}\, n \implies\ !!(\text{nil} = \text{nil}) * !!(n = 0)$. The (new) tactic `sep_inversion` applied to this goal leaves,

$$\frac{H_2 : l = \text{nil} \qquad H_3 : x = n}{!!(l = \text{nil})\ *\ !!(x = 0) \implies\ !!(\text{nil} = \text{nil})\ *\ !!(n = 0)}$$

Then the standard Coq `subst l x` leads to the sep_trivially solvable goal $!!(\text{nil} = \text{nil})\ *\ !!(n = 0) \implies\ !!(\text{nil} = \text{nil})\ *\ !!(n = 0)$.

6 Benchmark

The same reverse_list program has been proved twice with respect to identical axioms: once by Affeldt and Marti [1] using the style of unfolding the $*$ operator into its underlying semantics, and once as I have described in this paper. A rough comparison of their sizes (obtained from `wc`, not including comments or the definition of the reverse program itself and not including the general-purpose lemmas and tactics described in this paper) is,

	Lines	Words
unfolding	475	1,636
new tactics	200	795

That is, preserving the abstractions of separation logic using tactics adapted for that purpose makes proofs about half as large.

7 Conclusion

Separation logic uses Hoare triples $\{P\}C\{Q\}$ and sequents $P \implies Q$ which have a mixture of linear and classical conjuncts. While it is permissible to expand the linear conjunctions into their (classical) semantic meaning, proofs done this way are burdened with a tangle of heap-disjointness conditions. Therefore it is desirable to have lemmas and tactics that can manipulate the linear entailments directly.

On the other hand, one does not want to duplicate in linear logic all the lemma and tactic libraries that already exist (in Coq or Isabelle/HOL) for classical logic. One wants to prove the linear portions (involving memory access) one way, and the nonlinear portions (involving logic variables and local variables) another way.

What I have demonstrated is that a small set of lemmas will translate the nonlinear portions of the sequents into the native natural-deduction style of Coq or Isabelle/HOL. Each of these tactics is the analogue of a similar natural-deduction tactic in the underlying logic:

sep_trivial Associative-commutative $*$-rearrangement of impure conjuncts; drop, duplicate, and reconstruct pure conjuncts.

Forward Move past an assignment, load, or store, provided that the assigned variable does not appear free in the precondition.

assert_subst Perform a substitution using an equation appearing in the precondition.

assert_rewrite Perform a single rewrite using an equation appearing in the precondition.

Exists_left Eliminate an existential in the precondition.

Exists_right Instantiate an existential in the the r.h.s. of a sequent.

extract_prop Move a !!proposition from the precondition to the natural-deduction hypotheses.

assert_apply Apply a (semi)linear lemma to a sequent.

sep_inversion Apply syntactic inversion to an inductive predicate found in the precondition.

One can then relate a data structure (such as a list cell) to a mathematical object (such as a list); build sequent-logic proofs about the data structure and natural-deduction proofs about the mathematical object; and use the new tactics to move between the two systems with a minimum of difficulty.

Acknowledgments. Sandrine Blazy, Damien Doligez, Xavier Leroy, and Francesco Zappa-Nardelli proved in Coq many of the lemmas that support the tactics described in this paper.

References

[1] Reynald Affeldt and Nicolas Marti. http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/seplog/example_reverse_list.v, 2005.

[2] Reynald Affeldt and Nicolas Marti. Towards formal verification of memory properties using separation logic. http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/seplog, 2005.

[3] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Verification of the heap manager of an operating system using separation logic. In *SPACE 06: Third workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, January 2006.

[4] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic*, pages 1–19, September 2001. LNCS 2142.

[5] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

Appendix

The inductive *contents* predicate given in section 5.3 was defined so that (a) it has no free variables and (b) it uses as much as possible the connectives of the separation logic. However, the tactical framework described in this paper is robust enough to accommodate other choices. For example, the predicate contentsA relates a list l to an *expression* rather than an integer literal; the predicate contentsB uses "native Coq" for quantifications over h, t, ptr and the $l = h{::}t$ hypothesis. With either one, the proof of reverse_list can be completed without too much difficulty.

```
Inductive contentsA: list Z -> expr -> assert :=
 | contentsA_nil: forall l x,
    !!(l = nil) ** !(x == int_e 0)
    ==> contentsA l x
 | contentsA_cons: forall l x,
    (Exists h, Exists t, Exists ptr,
      !! (l = h::t) **
      (x +e int_e data|-> int_e h) **
      (x +e int_e next |-> ptr) **
      ! (x =/= int_e 0) **
      contentsA t  ptr)
    ==> contentsA l x.
Inductive contentsB: list Z -> expr -> assert :=
 | contentsB_nil: forall l x,
   l = nil ->
   !(x == int_e 0) ==> contentsB l x
 | contentsB_cons: forall l x h t ptr,
   l = h::t ->
   ((x +e int_e data|-> int_e h) **
    (x +e int_e next |-> ptr) **
    ! (x =/= int_e 0) **
    contentsB t  ptr
    ==> contentsB l x).
```

Valid predicates. The tactics for separation logic work over *valid* predicates, that is, those that are extensional over equivalent expressions. All of the primitive predicates are valid, and proofs of their validity are entered in a Coq "hint database" for use by the tactics. The contents relation on lists and integers is not a predicate on expressions and therefore doesn't need to be proved extensional. But relations such as contentsA and contentsB are predicates on expressions, and therefore one must prove the following theorem by induction on l and then enter it in the hint database for use by the various tactics.

$$\forall l e_1 e_2.\, \text{contentsA}\, l\, e_1 * !(e_1 = e_2) \implies \text{contentsA}\, l\, e_2.$$