Practical Tactics for Verifying C Programs in Coq

Jingyuan Cao   Ming Fu   Xinyu Feng
University of Science and Technology of China
cjy91312@mail.ustc.edu.cn   fuming@ustc.edu.cn   xyfeng@ustc.edu.cn

Abstract

Proof automation is essential for large scale proof development such as OS kernel verification. An effective approach is to develop tactics and SMT solvers to automatically prove verification conditions. However, for complex systems, it is almost impossible to achieve fully automated verification and human interactions are unavoidable. So the key challenge here is, on the one hand, to reduce manual proofs as much as possible, and on the other hand, to provide user-friendly error messages when the automated verification fails, so that users could adjust specifications or the code accordingly, or to do part of the proofs manually.

In this paper we propose a set of practical tactics for verifying C programs in Coq, including both tactics for automatically proving separation logic assertions and ones for automatic verification condition generation. In particular, we develop special tactics for verifying programs manipulating singly-linked lists. Using our tactics we are able to verify several C programs with one-line proof script. Another key feature of our tactics is that, if the tactics fail, they allow users to easily locate problems causing the failure by looking into the remaining subgoals, which greatly improves the usability when human interaction is necessary.

Categories and Subject Descriptors F.3.1 [*Logic and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods

General Terms Languages, Tactics, Verification

Keywords Interactive Proof Assistants, Practical Tactics, Separation Logic, C Program Verification

1. Introduction

Verifying system software, such as operating system kernels, has long been recognized as an important but also extremely challenging task. On the one hand, system software provides fundamental support of user applications. Its correctness is the prerequisite for the reliability of the whole system. On the other hand, verification of full functional correctness is difficult since the specifications need to be written in expressive logic languages and the theorem proving problem for such languages is usually undecidable. The fact that most operating system kernels are implemented in C makes the problem even harder, due to the pointer manipulations in the low-level programs. In practice, one has to pay great efforts for realistic system software verification. For instance, it took the seL4 team [13] more than 20 person years and 200,000 lines of Isabelle scripts to verify a microkernel consisting of around 8,000 lines of C code.

Due to complicated behaviors of system software, it is impossible to achieve fully automated verification of functional correctness. We usually need to combine automated verification procedures with manual interactive proofs together to verify system software in proof assistant tools. The good automated verification support should be able to reduce manual proofs as much as possible, and to provide user-friendly error messages when the automated verification fails so that manual efforts can easily take it over. Recently there has been much work proposed to support automated verification of C-like pointer programs, but these tools fail to satisfy the above two requirements. The "mostly automated" tools (*e.g.*, Bedrock [8]) require users to provide both specifications in the right forms and some hint lemmas for automated reasoning. It is very difficult to get all of them right in one shot, and when their tactics fail there is little information for users to find where the problems are. There are also some less automatic but easier to use tactics [2, 6, 16] for verifying C programs using separation logic [17], but they do not reduce the manual proofs as much as possible. Users still need to do many interactive proofs using these tactics.

In this paper, we propose a set of practical tactics for verifying C programs in Coq. Our tactics have higher degree of automation than existing tactics [2, 6, 16]. They also provide more user-friendly error information than Bedrock [8]. When the tactics fail, instead of showing a gigantic unproven verification condition, they either stop at the corresponding Hoare judgment where the failure occurs, which provides the right context for users to find the problems in the specifications or the code, or stop with pure assertions as subgoals, whose proofs are independent of program state (thus we do not need the corresponding program context). This is achieved through the following ideas:

- We introduce a resource checking phase before directly applying Hoare logic rules to generate subgoals (verification conditions). The resource checking checks whether the memory required by the execution of the statement is available in the pre-condition. Although the verification conditions of the logic rules would enforce similar requirements, the resource checking is like a pre-processing phase so that we can stop with the Hoare judgment (instead of subgoals of verification conditions) if the checking fails. The checking does searching and matching for the variables used in the expressions to locate the required resources. If it fails, the information about missing resources is helpful for users to locate problems in the specifications and the code.

CPP '15, January 13–14, 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3300-9/15/01...$15.00.
http://dx.doi.org/10.1145/2676724.2693162

- Based on the pre-condition, we do symbolic execution of expressions in the current statement. Results of the symbolic execution are used to instantiate and generate the post-condition in the forward reasoning, which can be used to continue the next forward step. The symbolic execution allows us to go further in the forward reasoning and gives us extra degree of automation.
- We extend the verification procedure with domain-specific knowledge. For now we only support automatic unfolding and folding inductive assertions about singly-linked lists. Such extra knowledge already allows us to out-perform existing work (see Sections 6 and 7 for details). It is possible to generalize this approach to support other data structures (*e.g.*, trees and doubly-linked lists) and theories (*e.g.*, 32-bit integers). The idea looks similar to shape-analysis techniques, but we do not enforce special constraints over the assertion language here.

This paper is based on previous work on separation logic tactics, but makes the following new contributions:

- We propose a set of practical tactics for verifying C programs in Coq, including both tactics for automatically proving separation logic assertions and ones for automatic verification condition generation. In particular, we incorporate domain-specific knowledge on singly-linked lists in the verification process for automated reasoning of list manipulating programs.
- Our tactics allow users to easily locate problems causing the failure of the automated proof procedure according to the user-friendly messages and the remaining subgoals, which greatly improves the usability when human interaction is necessary.
- We are able to verify some simple list-manipulating C programs with one-line Coq proof script using our tactics, which outperforms most existing separation logic tactics. Coq implementation of the tactics is available online from:

  `http://staff.ustc.edu.cn/~xyfeng/ptvc/`

In the rest of the paper, we describe a simplified C programming language in Sec. 2. We present the separation logic assertions for reasoning C programs in Sec. 3. Then we discuss our tactics sep_auto for proving the derivation of separation logic assertions in Sec. 4. After that, we discuss our program logic and the hoare_forward tactic we have created to prove Hoare judgements in Sec. 5. Then we discuss the implementation and evaluation of our tactics in Sec. 6. Finally we discuss related work in Sec. 7 and conclude in Sec. 8.

2. The Langauge

We present the ideas of our work based on a simplified small language in Fig. 1, showing some key features of the C programming language. Our tactics actually work on a reasonably practical subset of C, which has been used to implement an executable operating system kernel (a variant of μC/OS-II [14]).

We use i for 32-bit integers and a for memory addresses (pointers). A value v is either undefined, a 32-bit word value or a pointer.

An expression e follows the syntax of the C programming language. It is either a value, a program variable, memory reference, deference or standard arithmetic or logical operations over expressions. A statement s is either an assignment statement, a sequence of statements, a branch statement, a loop statement, or a **skip** statement that does nothing. The **while**-loop is annotated with a loop invariant I, which is a logic assertion provided by the programmer for verification purpose. Syntax of assertions will be introduced in the next section.

$$
\begin{array}{rcll}
(\textit{Integer}) & i & \in & \textit{Int32} \\
(\textit{Var}) & \mathsf{x} & \in & \textit{Int32} \\
(\textit{Addr}) & a & \in & \textit{Int32} \\
(\textit{Value}) & v & ::= & \mathsf{Vundef} \mid \mathsf{Vint}(i) \mid \mathsf{Vptr}(a) \\
(\textit{Expr}) & e & ::= & v \mid \mathsf{x} \mid *e \mid \&e \mid e.i \mid e+e \\
 & & \mid & e=e \mid e \neq e \mid \ldots \\
(\textit{Stmts}) & s & ::= & e=e \mid s;s \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \\
 & & \mid & \textbf{while}\ [I]\ (e)\ s \mid \textbf{skip} \mid \ldots \\
(\textit{State}) & \sigma & ::= & (m,u) \\
(\textit{Memory}) & m & \in & \textit{Addr} \rightharpoonup \textit{Value} \\
(\textit{SymTable}) & u & \in & \textit{Var} \rightharpoonup \textit{Addr}
\end{array}
$$

Figure 1. The Syntax

$$
[\![e]\!]^r_\sigma \triangleq \begin{cases}
v & \text{if } e = v \\
v & \text{if } e = \mathsf{x} \wedge [\![\mathsf{x}]\!]^l_\sigma = a \wedge \sigma.m(a) = v \\
v & \text{if } e = *e' \wedge [\![e']\!]^l_\sigma = a \wedge \sigma.m(a) = v \\
\mathsf{Vptr}(a) & \text{if } e = \&e' \wedge [\![e']\!]^l_\sigma = \mathsf{Vptr}(a) \\
v & \text{if } e = e'.i \wedge [\![e']\!]^l_\sigma = \mathsf{Vptr}(a) \wedge \\
 & \quad \sigma.m(a+i) = v \\
v_1+v_2 & \text{if } e = e_1+e_2 \wedge [\![e_1]\!]^r_\sigma = v_1 \wedge [\![e_2]\!]^r_\sigma = v_2 \\
\ldots & \\
\bot & \text{otherwise}
\end{cases}
$$

$$
[\![e]\!]^l_\sigma \triangleq \begin{cases}
\mathsf{Vptr}(a) & \text{if } e = \mathsf{x} \wedge \sigma.u(\mathsf{x}) = a \\
\mathsf{Vptr}(a) & \text{if } e = *e' \wedge [\![e']\!]^r_\sigma = \mathsf{Vptr}(a) \\
\mathsf{Vptr}(a+i) & \text{if } e = e'.i \wedge [\![e']\!]^l_\sigma = \mathsf{Vptr}(a) \\
\bot & \text{otherwise}
\end{cases}
$$

Figure 2. The Evaluation of Expressions

A program state σ is a pair of a memory m and a symbol table u. The memory m is modelled as a partial function from addresses to values. The symbol table u is a partial mapping from program variables x to addresses. Note that we use the flat memory model for the simplified language to simplify the presentation. In our real implementation we use a block-based memory model, as in CompCert [15]. The difference is orthogonal with our tactics development.

Figure 2 gives semantic functions of expression evaluations. $[\![e]\!]^r_\sigma$ and $[\![e]\!]^l_\sigma$ evaluate the right and left values of the expression e in the context of the state σ. Small-step operational semantics of the language is formally defined in Coq.

Figure 3 shows a simple program doing in-place linked-list reversal. This example could be verified using the one-line proof script "**repeat** hoare_forward; sep_pure". We use it as a running example to demonstrate the ideas of tactics implementation.

3. Separation Logic Assertions

Separation logic [17] is an extension of Hoare logic to reason about pointer manipulating programs. We follow separation logic and give our assertion language in Fig. 4. Semantics of assertions are defined in Fig. 5.

ListRev $\triangleq$

```
$\{ \ \mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_ * \mathsf{t} \rightarrowtail \_ \ \}$
y=NULL;
while [...] (x ≠ NULL){
  t=(*x).next; \\next = 4
  (*x).next=y;
  y=x;
  x=t;
}
x=y;
$\{ \ \mathsf{list}(\mathsf{x}, \mathsf{rev}(L)) * \mathsf{y} \rightarrowtail \_ * \mathsf{t} \rightarrowtail \_ \ \}$
```

Figure 3. In-place Linked-list Reversal

$$
\begin{array}{llll}
(\mathit{Asrt}) & p & ::= & \mathsf{emp} \mid \mathsf{true} \mid \mathsf{false} \mid a \mapsto v \mid e = v \mid \mathsf{x}@a \\
 & & \mid & p * p \mid \mathsf{Ex}\, x.p \mid p \wedge p \mid p \vee p \mid \langle P \rangle
\end{array}
$$

Figure 4. The Syntax of Assertions

Assertion emp specifies a program state with an empty memory. Assertion $a \mapsto v$ specifies a program state which has a singleton memory with v stored at the location a. Assertion $e = v$ specifies states where e could be evaluated to a defined value v. Assertion $\mathsf{x}@a$ means that the address of x in the symbol table is a and the state has an empty memory. The separating conjunction $p_1 * p_2$ means p_1 and p_2 hold over disjoint part of a state. Here we use $f \perp g$ to mean the two finite partial mappings f and g have disjoint domains. The union of two disjoint states σ_1 and σ_2 is defined as $\sigma_1 \uplus \sigma_2$. Assertion $\langle P \rangle$ requires that P should hold and the state should satisfy emp. Here P is a Coq proposition, *i.e.*, we have P : **Prop** in Coq. We omit other assertions here, which are standard separation logic assertions.

In Fig. 5, we also define some useful assertions with the primitive assertions. $\mathsf{x} \rightarrowtail v$ means that the singleton memory has only one memory cell, whose address is stored in the variable x and the value in the memory cell is v. $\mathsf{x}.i \rightarrowtail v$ has a similar meaning except for the address offset i. $\mathsf{istrue}(e)$ means that the right value of e is not $\mathsf{Vint}(0)$ or NULL. $\mathsf{isfalse}(e)$ has the opposite meanings. Note that NULL is short for $\mathsf{Vptr}(0)$. Our tactics could also be used to verify C programs manipulating singly-linked lists. We inductively define $\mathsf{lseg}(a_1, a_2, L)$ to specify the linked-list segments with the value list L. $\mathsf{list}(\mathsf{x}, L)$ is used to specify the singly-linked list pointed by the head pointer x.

4. Tactics for Separation Logic Assertions

To write a tactic for automated reasoning about C program, we first need to implement tactics which are able to prove separation logic assertions. We implement a tactic sep_auto for such a purpose. This section will give the detailed explanation of our approach to implementing sep_auto with the Ltac language [9] provided by Coq [1].

4.1 The Ltac Language of Coq

We use the code written in Ltac language to present our algorithms directly. Before we demonstrate the algorithms of implementing our tactics, we first give a brief overview of some key features of Ltac, which are frequently used in this paper.

Coq's "match goal" construct provides pattern backtracking upon failure. All possible matchings of the goal against the given

$$
\begin{array}{lll}
\sigma \models \mathsf{emp} & \text{iff} & \sigma.m = \emptyset \\
\sigma \models \mathsf{true} & & \text{Always} \\
\sigma \models \mathsf{false} & & \text{Never} \\
\sigma \models a \mapsto v & \text{iff} & \sigma.m = \{a \rightsquigarrow v\} \\
\sigma \models e = v & \text{iff} & [\![e]\!]^r_\sigma = v \wedge v \neq \mathsf{Vundef} \\
\sigma \models \mathsf{x}@a & \text{iff} & \sigma.u(\mathsf{x}) = a \wedge \sigma \models \mathsf{emp}
\end{array}
$$

$$
(m_1, u_1) \uplus (m_2, u_2) \triangleq
\begin{cases}
(m_1 \cup m_2, u_1) & \text{if} \quad m_1 \perp m_2, u_1 = u_2 \\
\mathit{undefined} & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{lll}
f \perp g & \triangleq & \mathit{dom}(f) \cap \mathit{dom}(g) = \emptyset \\
\sigma \models p_1 * p_2 & \text{iff} & \exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \uplus \sigma_2 \wedge \\
 & & \qquad \sigma_1 \models p_1 \wedge \sigma_2 \models p_2 \\
\sigma \models \mathsf{Ex}\, x.p & \text{iff} & \exists x. \sigma \models p \\
\sigma \models p_1 \wedge p_2 & \text{iff} & \sigma \models p_1 \wedge \sigma \models p_2 \\
\sigma \models p_1 \vee p_2 & \text{iff} & \sigma \models p_1 \vee \sigma \models p_2 \\
\sigma \models \langle P \rangle & \text{iff} & P \wedge (\sigma \models \mathsf{emp}) \\
 & & \\
\mathsf{x} \rightarrowtail v & \triangleq & \mathsf{Ex}\, a.\mathsf{x}@a * a \mapsto v \\
\mathsf{x}.i \rightarrowtail v & \triangleq & \mathsf{Ex}\, a.\mathsf{x}@a * (a{+}i) \mapsto v \\
\mathsf{istrue}(e) & \triangleq & \mathsf{Ex}\, v.e{=}v * \langle v \neq \mathsf{Vint}(0) \wedge v \neq \mathsf{NULL} \rangle \\
\mathsf{isfalse}(e) & \triangleq & \mathsf{Ex}\, v.e{=}v * \langle v = \mathsf{Vint}(0) \vee v = \mathsf{NULL} \rangle \\
\mathsf{lseg}(a_1, a_2, \mathsf{nil}) & \triangleq & \langle a_1 = a_2 \rangle \\
\mathsf{lseg}(a_1, a_2, v :: L) & \triangleq & a_1 \mapsto v * \mathsf{Ex}\, a.(a_1{+}4) \mapsto \mathsf{Vptr}(a) \\
 & & \qquad * \mathsf{lseg}(a, a_2, L') \\
\mathsf{list}(\mathsf{x}, L) & \triangleq & \mathsf{Ex}\, a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) \\
p_1 \Rightarrow p_2 & \triangleq & \forall \sigma. \sigma \models p_1 \rightarrow \sigma \models p_2 \\
p_1 \Leftrightarrow p_2 & \triangleq & p_1 \Rightarrow p_2 \wedge p_2 \Rightarrow p_1
\end{array}
$$

Figure 5. The Semantics of Assertions

pattern are tried out in turn: if the tactic on the right hand side of the clause fails, then it is backtracked, and another instantiation is tried. For instance, we can use the "**match** *goal* ... **end**" construct to implement the following tactic similar to the assumption tactic:

Ltac *mytac* := **match** *goal* **with** $[H : \_ \vdash \_] \Rightarrow$ exact H **end**.

Here $[H : \_ \vdash \_]$ is a proof context pattern that can match any hypothesis and exact fails when the goal is not immediate from the argument. Because of the backtracking semantics of match goal, exact will be applied to every assumption until the goal is solved or no hypothesis matches.

In addition to the powerful "match goal" construct, the most fundamental combinator is the sequential composition denoted as "$t_1; t_2$". Its informal meaning is to apply t_2 to every subgoal produced by the execution of t_1 in the current proof context. Also there is a more general sequential composition that has the form of "$t; [t_1 \mid t_2 \mid \ldots \mid t_n]$". It means that the tactic t is applied and then t_i is applied to the i-th ($1 \leq i \leq n$) subgoal generated by t. It fails if t does not generate exactly n subgoals. When the number of the generated subgoals exceeds n, one can simply use "$t; [t_1 \mid t_2 \mid \ldots \mid t_n ..]$", which applies t_n to solve the $(n + j)$-th ($j \geq 0$) subgoals . The tactic "first $[t_1 \mid t_2 \mid \ldots \mid t_n]$" executes the tactics $t_1, t_2, \ldots, t_n$ one by one until a non-failing one is found; otherwise the whole first fails. The loop tactic "**repeat** t" applies t recursively to all the generated subgoals until it eventually fails. The recursion stops in a subgoal when the tactic has failed. We implement our tactics based on the primitive tactics and above language constructs provided by Ltac.

4.2 Assertion Derivation : sep_auto

The goal of sep_auto is to prove "$p \Rightarrow p'$". Here we require that both p and p' follow the syntax by removing the conjunction and disjunction construct from the syntax definition in Fig. 4. It means that both the conjunction and disjunction operators of assertions are disallowed to appear in p and p'. We give the main procedure of sep_auto as below. Its implementation depends on four useful tactics: sep_normal, sep_split, sep_cancel and sep_pure.

```
Ltac sep_auto :=
 1    intros;
 2    match goal with
 3      | $H : ?\sigma \models \_ \vdash ?\sigma \models \_ \Rightarrow$
 4        sep_normal_in $H$;
 5        repeat match $type\ of\ H$ with
            | $\_ \models$ Ex _._ $\Rightarrow$ destruct $H$ as [ ? $H$ ]
          end;
 6        sep_split_in $H$;
 7        subst;
 8        sep_normal;
 9        repeat match $goal$ with
            | $\vdash \_ \models$ Ex _._ $\Rightarrow$ eexists
          end;
10        sep_split; [ sep_cancel | sep_pure.. ]
    end.
```

For each hypothesis H, we first call sep_normal_in to normalize H (line 4), which lifts all the existential quantifications to the left side of H and simplifies H. Then we destruct all the existential variables from H (line 5). sep_split_in is called to find pure propositions in H and split them out (line 6), then we use the generated equations to do substitution and simplification (line 7). sep_split_in can be easily implemented with lemma L3 in Fig. 6.

For the goal, we do the similar things at lines 8, 9 and 10. Here the difference is that we use eexists to remove all the existential variables in the goal, then sep_cancel and sep_pure are able to prove non-pure and pure subgoals respectively, which are generated by executing sep_split over the goal. The key ideas of implementing sep_auto lie in sep_normal and sep_cancel, which we will discuss more in the following sections.

4.3 Assertion Normalization : sep_normal

We use sep_normal to do normalization for a given assertion p, and it transforms p into a normalized assertion p', which has the form of $\mathsf{Ex}\ x_1, \ldots, x_n.p''$, where p'' has the following features: (1) it does not contain any existential quantification; (2) it does not have emp in it, and has at most one true; (3) the components of p'' conform to the right association. For instance, sep_normal normalizes the following assertion p into p'.

$$p \triangleq (\mathsf{Ex}\ x.A\ x * \mathsf{true}) * (\mathsf{Ex}\ y.B\ y * \mathsf{true}) * \mathsf{emp} * (\mathsf{Ex}\ a'.a \mapsto v * a{+}4 \mapsto a' * \mathsf{lseg}(a', 0, L))$$

$$p' \triangleq \mathsf{Ex}\ x, y, a'.(A\ x * (B\ y * (a \mapsto v * (a{+}4 \mapsto a' * (\mathsf{lseg}(a', 0, L) * \mathsf{true})))))$$

$$\frac{\forall x.(A\ x \Rightarrow B\ x) \quad t : \mathsf{Type} \quad A, B : t \rightarrow Asrt}{\mathsf{Ex}\ x.A\ x \Rightarrow \mathsf{Ex}\ y.B\ y}\ \text{L 1}$$

$$\frac{t : \mathsf{Type} \quad A : t \rightarrow Asrt}{(\mathsf{Ex}\ x.A\ x) * p \Leftrightarrow \mathsf{Ex}\ x.(A\ x * p)}\ \text{L 2}$$

$$\frac{}{\sigma \models \langle P \rangle * p \Leftrightarrow (\sigma \models p) \wedge P}\ \text{L 3}$$

$$\frac{p_1 \Rightarrow p_2 \quad a_1 = a_2}{\mathsf{x}@a_1 * p_1 \Rightarrow \mathsf{x}@a_2 * p_2}\ \text{L 4}$$

$$\frac{p_1 \Rightarrow p_2 \quad v_1 = v_2}{a \mapsto v_1 * p_1 \Rightarrow a \mapsto v_2 * p_2}\ \text{L 5}$$

$$\frac{p_1 \Rightarrow p_2 \quad L_1 = L_2}{\mathsf{lseg}(a_1, a_2, L_1) * p_1 \Rightarrow \mathsf{lseg}(a_1, a_2, L_2) * p_2}\ \text{L 6}$$

$$\frac{p_2 \Rightarrow p_2'}{p_1 * p_2 \Rightarrow p_1 * p_2'}\ \text{L 7} \qquad \frac{}{p \Rightarrow \mathsf{lseg}(a, a, \mathsf{nil}) * p}\ \text{L 8}$$

$$\frac{p_1 \Rightarrow \mathsf{lseg}(a_2, a_3, L_3) * p_2 \quad L_1{+}{+}L_3 = L_2}{\mathsf{lseg}(a_1, a_2, L_1) * p_1 \Rightarrow \mathsf{lseg}(a_1, a_3, L_2) * p_2}\ \text{L 9}$$

Figure 6. Selected Lemmas for Assertions

The following code gives the implementation details for sep_normal.

```
Ltac sep_normal :=
 1  match goal with
 2    | $\vdash \_ \models$ Ex _._ $\Rightarrow$
 3      eapply L1;
          [ intros $H'$; sep_normal; exact $H'$ | idtac ]
 4    | $\vdash \_ \models ?p \Rightarrow$
 5      match find_Ex $p$ with
 6        | None $\Rightarrow$ sep_simpl
 7        | Some $?n \Rightarrow$ sep_lift $n$; apply L2; sep_normal
      end
  end.
```

The purpose of sep_normal is to lift all the existential variables to the left side of the goal and simplify the goal. If the goal has the form of $\mathsf{Ex}\ x.p$, we apply L1 (see Fig. 6) to normalize it by dealing with the inner p first (lines 2-3). If it does not begin with Ex, we use find_Ex to find the position of the existentially quantified part in p , then call sep_lift to move it to the left side of p and apply L2 to expand the scope of the existential variable (lines 4-5,7). sep_lift is implemented based on the associative and commutative laws of separation logic assertions. If there does not exist any existentially quantified part in p, which means all the existential variables have been moved to the left side of p, we use sep_simpl to simplify p by removing the redundant true and emp according to some obvious properties of separation logic assertions (line 6).

4.4 Spatial Assertion Elimination : sep_cancel

We implement sep_cancel as below, it is used to prove that a spatial assertion without existential quantifications implies another. Here spatial assertions do not contain any pure assertions like $\langle P \rangle$.

sep_cancel first calls cancel_same to eliminate the same parts in the premise and goal (line 1). If the goal still cannot be solved

$$\frac{H : \sigma \models (\mathsf{Ex}\ a_1.\mathsf{y}@a_1 * a_1 \mapsto \mathsf{NULL}) * (\mathsf{Ex}\ a, a'.(\mathsf{Ex}\ a_1.\mathsf{x}@a_1 * a_1 \mapsto \mathsf{Vptr}(a)) * a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L)) * \langle \mathsf{length}(L) \geq 2 \rangle}{\begin{array}{l}\sigma \models (\mathsf{Ex}\ a_x, L_x.(\mathsf{Ex}\ a_1.\mathsf{x}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_x)) * \mathsf{lseg}(a_x, 0, L_x)) * (\mathsf{Ex}\ a_y, L_y.(\mathsf{Ex}\ a_1.\mathsf{y}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_y)) * \mathsf{lseg}(a_y, 0, L_y)) * \\ \quad \langle L_x {+\!\!+} \mathsf{rev}(L_y) = v :: L \rangle * \langle \mathsf{length}(L) \geq 2 \rangle * \langle L \neq \mathsf{nil} \rangle\end{array}}$$

$$\Downarrow (1). \boxed{\mathsf{sep\_normal}\ \textbf{in}\ H}$$

$$\frac{H : \sigma \models \mathsf{Ex}\ a_1, a, a', a_2.\mathsf{y}@a_1 * a_1 \mapsto \mathsf{NULL} * \mathsf{x}@a_2 * a_2 \mapsto \mathsf{Vptr}(a) * a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L) * \langle \mathsf{length}(L) \geq 2 \rangle}{\begin{array}{l}\sigma \models (\mathsf{Ex}\ a_x, L_x.(\mathsf{Ex}\ a_1.\mathsf{x}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_x)) * \mathsf{lseg}(a_x, 0, L_x)) * (\mathsf{Ex}\ a_y, L_y.(\mathsf{Ex}\ a_1.\mathsf{y}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_y)) * \mathsf{lseg}(a_y, 0, L_y)) * \\ \quad \langle L_x {+\!\!+} \mathsf{rev}(L_y) = v :: L \rangle * \langle \mathsf{length}(L) \geq 2 \rangle * \langle L \neq \mathsf{nil} \rangle\end{array}}$$

$$\Downarrow (2). \boxed{\textbf{repeat}\ \mathsf{destruct}\ H;\ \mathsf{sep\_split}\ \textbf{in}\ H}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{y}@a_1 * a_1 \mapsto \mathsf{NULL} * \mathsf{x}@a_2 * a_2 \mapsto \mathsf{Vptr}(a) * a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L)}{\begin{array}{l}\sigma \models (\mathsf{Ex}\ a_x, L_x.(\mathsf{Ex}\ a_1.\mathsf{x}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_x)) * \mathsf{lseg}(a_x, 0, L_x)) * (\mathsf{Ex}\ a_y, L_y.(\mathsf{Ex}\ a_1.\mathsf{y}@a_1 * a_1 \mapsto \mathsf{Vptr}(a_y)) * \mathsf{lseg}(a_y, 0, L_y)) * \\ \quad \langle L_x {+\!\!+} \mathsf{rev}(L_y) = v :: L \rangle * \langle \mathsf{length}(L) \geq 2 \rangle * \langle L \neq \mathsf{nil} \rangle\end{array}}$$

$$\Downarrow (3). \boxed{\mathsf{sep\_normal};\ \textbf{repeat}\ \mathsf{eexists};\ \mathsf{sep\_split}}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{y}@a_1 * a_1 \mapsto \mathsf{NULL} * \mathsf{x}@a_2 * a_2 \mapsto \mathsf{Vptr}(a) * a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L)}{\begin{array}{ll}\mathsf{goal}\ 1 : & \sigma \models \mathsf{x}@?1 * ?1 \mapsto \mathsf{Vptr}(?2) * \mathsf{lseg}(?2, 0, ?3) * \mathsf{y}@?4 * ?4 \mapsto \mathsf{Vptr}(?5) * \mathsf{lseg}(?5, 0, ?6) \\ \mathsf{goal}\ 2,3,4 : & ?3 {+\!\!+} \mathsf{rev}(?6) = v :: L \qquad \mathsf{length}(L) \geq 2 \qquad \mathsf{length}(L) \neq 0\end{array}}$$

$$\Downarrow (4). \boxed{\mathsf{sep\_cancel}\ (\ \mathsf{line\ 1\ cancel\_same}\ )}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{lseg}(a, a', v :: \mathsf{nil}) * \mathsf{lseg}(a', 0, L)}{\begin{array}{ll}\mathsf{goal}\ 1 : & \sigma \models \mathsf{lseg}(a, 0, ?3) * \mathsf{lseg}(0, 0, ?6) \\ \mathsf{goal}\ 2,3,4 : & ?3 {+\!\!+} \mathsf{rev}(?6) = v :: L \qquad \mathsf{length}(L) \geq 2 \qquad \mathsf{length}(L) \neq 0\end{array}}$$

$$\Downarrow (5). \boxed{\mathsf{sep\_cancel}\ (\ \mathsf{line\ 8}\ )}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{lseg}(a', 0, L)}{\begin{array}{ll}\mathsf{goal}\ 1 : & \sigma \models \mathsf{lseg}(a', 0, ?7) * \mathsf{lseg}(0, 0, ?6) \\ \mathsf{goal}\ 2,3,4 : & (v :: \mathsf{nil} {+\!\!+} ?7) {+\!\!+} \mathsf{rev}(?6) = v :: L \qquad \mathsf{length}(L) \geq 2 \qquad \mathsf{length}(L) \neq 0\end{array}}$$

$$\Downarrow (6). \boxed{\mathsf{sep\_cancel}\ (\ \mathsf{line\ 1\ cancel\_same}\ )}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{emp}}{\begin{array}{ll}\mathsf{goal}\ 1 : & \sigma \models \mathsf{lseg}(0, 0, ?6) \\ \mathsf{goal}\ 2,3,4 : & (v :: \mathsf{nil} {+\!\!+} L) {+\!\!+} \mathsf{rev}(?6) = v :: L \qquad \mathsf{length}(L) \geq 2 \qquad \mathsf{length}(L) \neq 0\end{array}}$$

$$\Downarrow (7). \boxed{\mathsf{sep\_cancel}\ (\ \mathsf{line\ 9\ cancel\_emp\_lseg}\ )}$$

$$\frac{H1 : \mathsf{length}(L) \geq 2 \qquad H : \sigma \models \mathsf{lseg}(a, a', v :: \mathsf{nil}) * \mathsf{x}@a_2 * a_2 \mapsto \mathsf{Vptr}(a) * \mathsf{y}@a_1 * a_1 \mapsto \mathsf{NULL} * \mathsf{lseg}(a', 0, L)}{\mathsf{goal}\ 1,2,3 : \quad (v :: \mathsf{nil} {+\!\!+} L) {+\!\!+} \mathsf{rev}(\mathsf{nil}) = v :: L \qquad \mathsf{length}(L) \geq 2 \qquad \mathsf{length}(L) \neq 0}$$

$$\Downarrow (8). \boxed{\mathsf{sep\_pure}}$$

No more subgoals.

Figure 7. A Step-by-Step Example using sep_auto

it applies lemma L9 (see Fig. 6) to subtract the sub-list segment in the goal and try sep_cancel again until the goal is solved.

```
Ltac sep_cancel :=
 1    cancel_same;
 2    fold_node_in $H$; fold_node;
 3    match $goal$ with
 4      | $H : ?\sigma \models ?p \vdash ?\sigma \models ?q \Rightarrow$
 5        match search_match_lseg $p$ $q$ with
 6          | Some $(?m, ?n) \Rightarrow$
 7            sep_lift $m$; sep_lift_in $H$ $n$;
 8            try solve [ apply L8; sep_cancel ];
 9            try solve [ gen $H$; apply L9;
                         intros; [ sep_cancel | sep_pure.. ] ]
10          | None ⇒ cancel_emp_lseg
          end
      end.
```

Note that cancel_same may not solve the whole goal when list segments appear in the hypothesis or the goal. For example, suppose that the hypothesis is "$\sigma \models \mathsf{lseg}(a, a_1, L) * \mathsf{lseg}(a_1, a', L')$", and the goal is "$\sigma \models \mathsf{lseg}(a, a', L{+\!\!+}L')$". Because there are no identical parts found by search_match in the hypothesis and the goal (line 14), cancel_same does nothing. The remaining goal needs to be proved by tactics from line 2 to line 10. Here we use another function search_match_lseg to find the matching components like $\mathsf{lseg}(a_1, a_2, L_1)$ and $\mathsf{lseg}(a_1, a_2', L_2)$, whose tail pointers a_2 and a_2' do not need to be identical. The positions of $\mathsf{lseg}(a, a_1, L)$ and $\mathsf{lseg}(a, a', L{+\!\!+}L')$ can be returned (line 4), then we lift them to the left side and subtract $\mathsf{lseg}(a, a_1, L)$ from $\mathsf{lseg}(a, a', L{+\!\!+}L')$ by applying L9. Finally the remaining goal is exactly the same as the hypothesis and could be proved by cancel_same, and the pure subgoal generated by applying L9 can be trivially solved now.

Note that we need to consider how to deal with the empty list segment, before applying L9 to subtract the list segment from the goal. We try to eliminate the list segment using L8 (line 7), which means if the list segment in the goal is an empty one we can trivially remove it. When search_match_lseg cannot find any matching components, we try to solve the remaining goals by eliminating the empty list segment both in the hypothesis and the goals using cancel_emp_lseg (line 9), which can be easily implemented with L8 and we omit it here.

We implement cancel_same like this: if the goal is true or the same as the hypothesis H, it can be trivially solved (lines 11-12), otherwise we use search_match to find the assertions that match each other in the hypothesis and the goal (line 14), then lift them to the left side of the assertion (line 16) and cancel both of them according to the lemmas L4, L5, L6 and L7 (lines 18-21). Here search_match searches the hypothesis and the goal, and returns the positions of matching components, which have the following forms: $\mathsf{x}@a_1$ and $\mathsf{x}@a_2$, $a \mapsto v_1$ and $a \mapsto v_2$, $\mathsf{lseg}(a_1, a_2, L_1)$ and $\mathsf{lseg}(a_1, a_2, L_2)$, and any other components that are exactly the same. We do this routine repetitively to eliminate all the identical parts in the hypothesis and the goal. We give the code for cancel_same as below, in which gen is short for "generalize dependent".

```
Ltac cancel_same :=
  match goal with
    | ⊢ ?σ ⊨ true ⇒ simpl; auto
    | H : ?σ ⊨ ?p ⊢ ?σ ⊨ ?q ⇒
      try solve [ exact H ];
      match search_match p q with
        | Some (?m, ?n) ⇒
          sep_lift m; sep_lift_in H n;
          match goal with
            | ⊢ _ ⊨ _@_ ∗ _ ⇒
              gen H; apply L4; intros;
                              [ cancel_same | sep_pure.. ]
            | ⊢ _ ⊨ _↦_ ∗ _ ⇒
              gen H; apply L5; intros;
                              [ cancel_same | sep_pure.. ]
            | ⊢ _ ⊨ lseg(_, _, _) ∗ _ ⇒
              gen H; apply L6; intros;
                              [ cancel_same | sep_pure.. ]
            | _ ⇒ gen H; apply L7; cancel_same
          end
        | None ⇒ idtac
      end
  end.
```

4.5 An Example using sep_auto.

Figure 7 presents a detailed procedure for using sep_auto to prove a non-trivial derivation of separation logic assertions. It shows that how our sep_auto works. After running the normalization and preprocess steps (1)(2) and (3), all the existential variables and pure propositions are split out in the hypothesis, while all the existential variables turns to un-instantiated variables (?n) that need to be instantiated in the future. Then sep_cancel attempts to cancel the common parts at steps (4)(5)(6) and (7). The procedure of cancelation will instantiate variables "?n" and some pure propositions are generated as well. After that we use sep_pure to solve the left pure goals at step (8).

5. Tactics for Judgements of Program Logics

In this section, we present a tactic hoare_forward which is designed to build automated proofs for generating verification conditions from Hoare judgements. The purpose of hoare_forward is to make one forward-step proof for Hoare judgements, and the tactic can be easily constructed like "**repeat** hoare_forward; sep_pure" to automatically prove the entire Hoare judgement. Here sep_pure is an extensible tactic for solving the pure subgoals generated by running hoare_forward. When the repeated calls to hoare_forward fail to prove the goal, we are able to use hoare_forward to do step-by-step reasoning, which is helpful for us to find the places where proofs fail. If hoare_forward fails without doing anything, we need to check if the resources (*i.e.*, memory) specified by the pre-condition is enough for the current statement. When the repeated execution of hoare_forward succeeds, the remaining subgoals will be proved by sep_pure. Note that we do not guarantee that sep_pure can solve all the remaining subgoals. If it fails we need to fix the pure parts of the specifications or the code to make sure that those pure subgoals can be solved. If the subgoals are indeed valid, users may also

$$\frac{}{\vdash \{p\}\mathbf{skip}\{p\}}\ \textsc{skip} \qquad \frac{\vdash \{p_1\}s_1\{p_2\} \quad \vdash \{p_2\}s_2\{p_3\}}{\vdash \{p_1\}s_1; s_2\{p_3\}}\ \textsc{seq}$$

$$\frac{\vdash \{p_1\}s\{p_2'\} \quad p_2' \Rightarrow p_2}{\vdash \{p_1\}s\{p_2\}}\ \textsc{forward}$$

$$\frac{\vdash \{p_1'\}s\{p_2\} \quad p_1 \Rightarrow p_1'}{\vdash \{p_1\}s\{p_2\}}\ \textsc{backward}$$

$$\frac{\vdash \{p_1\}s\{q\} \quad \vdash \{p_2\}s\{q\}}{\vdash \{p_1 \vee p_2\}s\{q\}}\ \textsc{disj}$$

$$\frac{\begin{array}{c}\vdash \{p_1 \wedge \mathsf{istrue}(e)\}s_1\{p_2\} \quad \vdash \{p_1 \wedge \mathsf{isfalse}(e)\}s_2\{p_2'\} \\ p_1 \Rightarrow \mathsf{Ex}\ v.e = v\end{array}}{\vdash \{p_1\}\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\{p_2 \vee p_2'\}}\ \textsc{if}$$

$$\frac{\vdash \{I \wedge \mathsf{istrue}(e)\}s\{I\} \quad I \Rightarrow \mathsf{Ex}\ v.e = v}{\vdash \{I\}\mathbf{while}\ [I]\ (e)\ s\{I \wedge \mathsf{isfalse}(e)\}}\ \textsc{while}$$

$$\frac{\vdash \{p_1\}s\{p_2\}}{\vdash \{\mathsf{Ex}\ x.p_1\}s\{\mathsf{Ex}\ x.p_2\}}\ \textsc{exist}$$

$$\frac{p \Rightarrow \&e_1 = \mathsf{Vptr}(a) \quad p \Leftrightarrow a \mapsto v * p' \quad p \Rightarrow e_2 = v'}{\vdash \{p\}e_1 = e_2\{a \mapsto v' * p'\}}\ \textsc{assign}$$

Figure 8. Inference Rules

extend the tactic sep_pure to make it more powerful to prove the subgoals automatically, or simply prove them manually.

5.1 The hoare_forward Tactic

Figure 9 presents the implementation of hoare_forward, which is used to make a forward step for proving the judgment $\vdash \{p\}s\{q\}$. To apply it, we make the same requirement as sep_auto, both the conjunction and disjunction operators are disallowed in the pre- and post-conditions given by users. We only deal with the conjunction and disjunction operators produced by applying the inference rules. We implement hoare_forward in the following three steps:

1. We check and unfold the pre-condition to make sure that the resource specified by the pre-condition is enough for evaluating the expressions whose results are required by the inference rules shown in Fig. 8.
2. We repetitively apply the sequential composition rule (the SEQ rule in Fig. 8) to decompose the entire Hoare judgement into small pieces, and we only deal with the first one.
3. We apply the proper inference rule to make a forward step for proving the judgement of the first statement which has an uninstantiated post-condition (*i.e.*, p_2 in the SEQ rule), then we do symbolic execution for expressions in the statement to obtain the values which are used to instantiate the post-condition.

In the first step (lines 2-4), the current pre-condition may have a disjunction operator introduced by the last post condition when applying the IF rule, so we apply the DISJ rule to continue the forward step with two different pre-conditions (line 2). If there is no disjunction operator in the pre-condition, we first call find_first_inv_exprs to return a pair consisting of an optional assertion and a list of expressions. The assertion is the annotated loop invariant if the s-

```
Ltac hoare_forward :=
 1  match goal with
 2    | ⊢ (⊢ {_ ∨ _}_{_}) ⇒ eapply DISJ; hoare_forward
 3    | ⊢ (⊢ {_}?s{_}) ⇒
 4      let l := find_first_inv_exprs s in hoare_unfold l;
 5               hoare_forward_first
    end.

Ltac hoare_forward_first :=
 6   match goal with
 7     | ⊢ (⊢ {_}_;_{_}) ⇒
 8       eapply SEQ; [ hoare_forward_first | idtac ]
 9     | ⊢ (⊢ {_}_{_}) ⇒
10       eapply FORWARD; intros;
11         [ hoare_forward_stmt
12           | intros H; first [ exact H | sep_auto ] ]
     end.

Ltac hoare_forward_stmt :=
13   match goal with
14     | ⊢ (⊢ {_}if _ then _ else _{_}) ⇒
15       eapply IF;
16         [ eapply BACKWARD;
17           [ idtac | intros H;
                          sep_conj_to_star_in H; exact H ]
18         | eapply BACKWARD;
19           [ idtac | intros H;
                          sep_conj_to_star_in H; exact H ]
20         | symbolic_exe ]
21     | ⊢ (⊢ {_}while [_] (_) _{_}) ⇒
22       eapply FORWARD;
23         [ eapply WHILE;
24           [ eapply BACKWARD;
25             [ idtac | intros H;
                          sep_conj_to_star_in H; exact H ]
26           | symbolic_exe ]
27         | intros H; sep_conj_to_star_in H; exact H ]
28     | ⊢ (⊢ {_}skip{_}) ⇒ apply SKIP
29     | ⊢ (⊢ {_}_=_{_}) ⇒
30       repeat apply EXIST; eapply ASSIGN;
31         [ symbolic_exe
32         | split; intros; cancel_same
33         | symbolic_exe ]
     end.
```

Figure 9. The Tactic hoare_forward for Proving "$\vdash \{p\}s\{q\}$"

tatement is a loop (None otherwise). The list of expressions are those appear in the first statement. Then hoare_unfold is used to do resource check by inspecting the pre-condition and the aforementioned loop invariant (if any) to ensure there is enough resource for computing the list of expressions (line 4). It may also unfold lseg (if any) in the pre-condition or the loop invariant before doing the resource check. After that, we call hoare_forward_first to do the second step, which repetitively apply the SEQ rule to get the first Hoare judgment we need to prove. The judgement has the form of $\vdash \{p\}s\{?q\}$, in which the statement s does not contain the sequential connector ";" and the post condition is uninstantiated. Then we call hoare_forward_stmt to finish the third step (line 11), and the post condition is instantiated for the next hoare_forward step.

The tactic hoare_forward_stmt tries to apply the proper inference rule by matching the current goal, and the generated side-conditions like "$p \Rightarrow e =?v$" are solved by the tactic symbolic_exe (lines 20, 26, 31 and 33), which does symbolic execution to get the value of expressions based on the resource specified by p. The returned value is used to instantiate "$?v$", and some related properties (*eg.* v is not Vundef) for the values are guaranteed by generating some pure goals. These pure goals remain to be proven after repetitively doing hoare_forward. Here applying the IF and WHILE rules will introduce the conjunction operators in the pre-condition of the subgoal. As mentioned before, since our hoare_forward tactic requires that the pre-conditions must have the form of $p_1 * p_2 * \ldots * p_n$, we need to convert the pre-condition into the required form to allow subsequent calls to hoare_forward. We use a tactic sep_conj_to_star_in to convert the conjunction operators into the separating conjunction operator (lines 17, 19, 25 and 27). sep_conj_to_star_in can be easily implemented with the following two lemmas, and their premises "$p \Rightarrow e =?v$" can be proven by symbolic_exe.

$$\frac{p \Rightarrow e = v}{p \wedge \mathsf{istrue}(e) \Rightarrow p * \langle v \neq \mathsf{Vint}(0) \wedge v \neq \mathsf{NULL}\rangle}$$

$$\frac{p \Rightarrow e = v}{p \wedge \mathsf{isfalse}(e) \Rightarrow p * \langle v = \mathsf{Vint}(0) \vee v = \mathsf{NULL}\rangle}$$

In the following paragraphs, we will explain more about how our hoare_unfold and symbolic_exe work.

5.2 Checking Resource and Unfolding Lists – hoare_unfold

Figure 10 presents the main routine of implementing hoare_unfold, which takes an optional assertion a and a list of expressions l as its input. When a is None, hoare_unfold first applies BACKWARD rule, then use sep_normal_in to normalize the pre-condition (lines 9 and 11). The hypothesis H is converted into an equivalent one that has the form of "$\mathsf{Ex}\ x_1, \ldots, x_n.p$", then we use sep_unfold_in H e to get a new pre-condition that has the resource for calculating e (line 12). When a is Some I, hoare_unfold first does hoare_unfold (None, l) to check the current pre-condition has enough resource for the expressions in l, then uses BACKWARD to convert the pre-condition into the loop invariant I and uses sep_auto to prove that the original pre-condition implies the new pre-condition I (lines 4-5). Then hoare_unfold tries to unfold the list segments with I to make sure that I provides enough resource for calculating the expressions in l (lines 7-13).

Since sep_unfold_in H e always takes a normalized H as its input, we can repetitively apply L1 (lines 15-18) to make sure that check_resource uses an assertion p without any existential quantifications (line 19). If check_resource returns Some v, it means that p has the resource for e and sep_unfold_in succeeds with idtac (line 21), otherwise it will try to unfold the list segments in the pre-conditions to obtain unfolded list nodes for calculating e (lines 22-30). For example, suppose we have the pre-condition "$\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L)$" and we need to calculate the value of $(*\mathsf{x}).4$. Then check_resource cannot find the required memory "$a{+}4 \mapsto \_$", and it fails to pass the resource check procedure. However, "$\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L)$" does contain the resource "$a{+}4 \mapsto \_$" if $a \neq 0$. We can assume $a \neq 0$ and apply the UNFOLD lemma below:

$$\frac{a \neq 0 \quad \sigma \models \mathsf{lseg}(a, 0, L) * p}{\sigma \models \mathsf{Ex}\ v, L'.\langle L = v :: L'\rangle * \mathsf{lseg}(a, 0, L) * p}\ \text{UNFOLD}$$

It unfolds the pre-condition into the following one, which contains the resource demanded by $(*\mathsf{x}).4$.

$$\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{Ex}\ v, L', a'.\langle L = v :: L'\rangle * \\ a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L')$$

```
Ltac hoare_unfold $x$ :=
 1   match $x$ with
 2     | (Some $?I, ?l$) ⇒
 3       hoare_unfold (None, $l$);
 4         [ apply BACKWARD with ($p'_1 := I$);
 5           [ hoare_unfold (None, $l$) | sep_auto ]
 6         | idtac.. ]
 7     | (None, ($?e_1 + ?e_2$) :: $?l$) ⇒
                          hoare_unfold (None, $e_1 :: e_2 :: l$)
 8     | (None, $?e :: ?l$) ⇒
 9       eapply BACKWARD;
10         [ idtac
11         | intros $H$; sep_normal_in $H$;
12           sep_unfold_in $H$ $e$; [ exact $H$ | idtac.. ] ];
13       [ hoare_unfold (None, $l$) | idtac.. ];
     end.

Ltac sep_unfold_in $H$ $e$ :=
14   match $type\ of\ H$ with
15     | _ ⊨ Ex _._ ⇒
16       eapply L1 in $H$;
17         [ idtac
18         | intros $H'$; sep_unfold_in $H'$ $e$;
                              [ exact $H'$ | idtac.. ] ]
19     | _ ⊨ $?p$ ⇒
20       match check_resource $p$ $e$ with
21         | Some _ ⇒ idtac
22         | None ⇒
23           match $e$ with
24             | $?e'$._ ⇒
25               match check_resource $p$ ($\&e'$) with
26                 | None ⇒
27                   sep_unfold_in $H$ ($\&e'$);
28                   [ sep_normal_in $H$;
                           sep_unfold_in $H$ $e$ | idtac.. ]
29                 | Some Vptr($?a$) ⇒
                           sep_unfold_lseg_in $H$ $a$
30                 | Some _ ⇒ fail
                   end
31             | _ ⇒ fail
           end  end  end.

Ltac sep_unfold_lseg_in $H$ $a$ :=
32   match find_math $p$ lseg($a$, _, _) with
33     | None ⇒ fail
34     | Some $?n$ ⇒
35       sep_lift_in $H$ $n$; apply UNFOLD in $H$;
36        [ sep_rewrite_pure_in $H$; unfolds_lseg | idtac ]
     end
```

Figure 10. The Tactc hoare_unfold for Unfolding Pre-conditions

To handle the above situation, we try to unfold the list segments when the resource checking fails (lines 22-30). If check_resource returns None for expression $e.i$, we continue to call check_resource on $\&e$ until it returns Some Vptr(a), then we call sep_unfold_lseg_in to unfold a head node of the list segment starting from a (line 29). sep_unfold_lseg_in applies UNFOLD to the hypothesis and use sep_rewrite_pure_in to rewrite the generated equations in H, then unfold the list segment by the definition of lseg (line 36).

Below we give an example to show how our hoare_unfold works. Suppose we want to prove the Hoare judgement below:

$$\begin{array}{l} \vdash \{\ \mathsf{Ex}\ a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle \mathsf{length}(L) \geq 2 \rangle\ \} \\ \quad \mathsf{x} = (*((*\mathsf{x}).\mathsf{next})).\mathsf{next} \\ \quad \{...\} \end{array}$$

We have to unfold the list segment twice. hoare_unfold will unfold the pre-condition in the goal to make sure that the expressions in the assignment statement "x := $(*((*\mathsf{x}).\mathsf{next})).\mathsf{next}$" (next = 4) be able to pass the resource check. Here we know length$(L) \geq 2$, and it is safe to unfold it twice. We first apply the BACKWARD rule to generate the following subgoal, then we are able transform the assertion into an equivalent one, which is used to instantiate ?1.

$$\frac{H : \sigma \models \mathsf{Ex}\ a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle \mathsf{length}(L) \geq 2 \rangle}{\sigma \models ?1}$$

By applying L1 we only need to construct an equivalent assertion according to $H1$.

$$\frac{H1 : \sigma \models \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle \mathsf{length}(L) \geq 2 \rangle}{\sigma \models ?2}$$

Obviously, $H1$ does not contain the resource demanded by $(*((*\mathsf{x}).\mathsf{next})).\mathsf{next}$, thus check_resource fails. Then sep_unfold_in will try to unfold $(*\mathsf{x}).\mathsf{next}$ first. $H1$ still does not contain the resource for $(*\mathsf{x}).\mathsf{next}$ and sep_unfold_in will try to unfold x. Now $H1$ has the resource of x and the value of x is Vptr(a), so sep_unfold_in will call sep_unfold_lseg_in, which tries to lift lseg$(a, 0, L)$ to the left side and applies UNFOLD for unfolding. After that $H1$ contains a pure assertion $L = v_1 :: L_1$. We use sep_rewrite_pure_in to substitute L in $H1$ and unfold lseg$(a, 0, v_1 :: L_1)$ by the definition of lseg, and then call sep_normal_in to normalize $H1$ as below :

$$\frac{\begin{array}{l} H1 : \sigma \models \mathsf{Ex}\ v_1, L_1, a_1.\langle L = v_1 :: L_1 \rangle * a \mapsto v_1 * \\ \qquad a{+}4 \mapsto \mathsf{Vptr}(a_1) * \mathsf{lseg}(a_1, 0, L_1) * \\ \qquad \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \langle \mathsf{length}(v_1 :: L_1) \geq 2 \rangle \end{array}}{\sigma \models ?2}$$

Since we successfully unfold $(*\mathsf{x}).\mathsf{next}$ with the current pre-conditions, we try to unfold $(*((*\mathsf{x}).\mathsf{next})).\mathsf{next}$ under $H1$ as before. Some unknown existential variables will be instantiated and we get the new pre-condition in $H2$:

$$\frac{\begin{array}{l} H2 : \sigma \models \langle L = v_1 :: L_1 \rangle * a \mapsto v_1 * a{+}4 \mapsto \mathsf{Vptr}(a_1) * \\ \qquad \mathsf{lseg}(a_1, 0, L_1) * \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \langle \mathsf{length}(v_1 :: L_1) \geq 2 \rangle \end{array}}{\sigma \models ?3}$$

Now the pre-condition contains the resource of $(*\mathsf{x}).\mathsf{next}$ and its value is Vptr(a_1). Then sep_unfold_in finds lseg$(a_1, 0, L_1)$ and continues to do the same thing as before. Finally it turns $H2$ into $H3$ that has the pre-condition we want:

$$\frac{\begin{array}{l} H3 : \sigma \models \mathsf{Ex}\ v_2, L_2, a_2.\langle L_1 = v_2 :: L_2 \rangle * a_1 \mapsto v_2 * \\ \qquad a_1{+}4 \mapsto \mathsf{Vptr}(a_2) * \mathsf{lseg}(a_2, 0, L_2) * \\ \qquad \langle L = v_1 :: v_2 :: L_2 \rangle * a \mapsto v_1 * a{+}4 \mapsto \mathsf{Vptr}(a_1) * \\ \qquad \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \langle \mathsf{length}(v_1 :: v_2 :: L_2) \geq 2 \rangle \end{array}}{\sigma \models ?3}$$

Finally we instantiate ?3, ?2 and ?1 and turn the original pre-condition into the following:

$$\begin{array}{l} \vdash \left\{ \begin{array}{l} \mathsf{Ex}\ a, v_1, L_1, a_1, v_2, L_2, a_2.\langle L_1 = v_2 :: L_2 \rangle * \\ a_1 \mapsto v_2 * a_1{+}4 \mapsto \mathsf{Vptr}(a_2) * \mathsf{lseg}(a_2, 0, L_2) * \\ \langle L = v_1 :: v_2 :: L_2 \rangle * a \mapsto v_1 * a{+}4 \mapsto \mathsf{Vptr}(a_1) * \\ \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \langle \mathsf{length}(v_1 :: v_2 :: L_2) \geq 2 \rangle \end{array} \right\} \\ \quad \mathsf{x} = (*((*\mathsf{x}).\mathsf{next})).\mathsf{next} \\ \quad \{...\} \end{array}$$

```
Ltac symbolic_exe :=
1   intros;
2   match goal with
3     | H : ?σ ⊨ _ ⊢ ?σ ⊨ _ ⇒
4       sep_normal in H;
5       repeat match type of H with
            | _ ⊨ Ex _._ ⇒ destruct H as [ ? H ]
          end;
6       sep_split in H; subst;
7       repeat match goal with
            | ⊢ _ ⊨ Ex _._ ⇒ eexists
          end;
8       sep_get_rv
    end.

Ltac sep_get_rv :=
9    match goal with
10     | H : ?σ ⊨ _ ⊢ ?σ ⊨ (?e1+?e2) = _ ⇒
         eapply SE1; [ sep_get_rv | sep_get_rv | idtac ]
11     | H : ?σ ⊨ _ ⊢ ?σ ⊨ ∗?e = _ ⇒
         eapply SE2; [ sep_get_rv | sep_auto | idtac ]
12     | H : ?σ ⊨ _ ⊢ ?σ ⊨ ?e.?i = _ ⇒
         eapply SE3; [ sep_get_rv | sep_auto | idtac ]
13     | H : ?σ ⊨ _ ⊢ ?σ ⊨ &(∗?e) = _ ⇒
         eapply SE4; sep_get_rv
14     | H : ?σ ⊨ _ ⊢ ?σ ⊨ &(?x) = _ ⇒
         eapply SE5; sep_auto
15     | H : ?σ ⊨ _ ⊢ ?σ ⊨ ?x = _ ⇒
         eapply SE6; [ sep_get_rv | sep_auto | idtac ]
16     | H : ?σ ⊨ _ ⊢ ?σ ⊨ NULL = _ ⇒
         apply SE7
   end.
```

Figure 11. sep_get_rv and symbolic_exe

$$\frac{\sigma \models e_1 = v_1 \quad \sigma \models e_2 = v_2 \quad v_1 + v_2 \neq \mathsf{Vundef}}{\sigma \models (e_1 + e_2) = v_1 + v_2}\ \textsc{se}\ 1$$

$$\frac{\sigma \models e = \mathsf{Vptr}(a) \quad \sigma \models a \mapsto v * p \quad v \neq \mathsf{Vundef}}{\sigma \models *e = v}\ \textsc{se}\ 2$$

$$\frac{\sigma \models \&e = \mathsf{Vptr}(a) \quad \sigma \models a + i \mapsto v * p \quad v \neq \mathsf{Vundef}}{\sigma \models e.i = v}\ \textsc{se}\ 3$$

$$\frac{\sigma \models e = v}{\sigma \models \&(*e) = v}\ \textsc{se}\ 4 \qquad \frac{\sigma \models \mathsf{x}@a * p}{\sigma \models \&\mathsf{x} = \mathsf{Vptr}(a)}\ \textsc{se}\ 5$$

$$\frac{\sigma \models \&\mathsf{x} = \mathsf{Vptr}(a) \quad \sigma \models a \mapsto v * p \quad v \neq \mathsf{Vundef}}{\sigma \models \mathsf{x} = v}\ \textsc{se}\ 6$$

$$\frac{}{\sigma \models \mathsf{NULL} = \mathsf{Vptr(0)}}\ \textsc{se}\ 7$$

Figure 12. Symbolic Execution Rules

In this example, since hoare_unfold successfully unfolds the list segment twice, hoare_forward can continue with the assignment statement because all the resources are ready.

5.3 Symbolic Execution for Expressions – symbolic_exe

We use symbolic_exe to prove "$p \Rightarrow e = ?v$" by instantiating $?v$ with the value obtained from symbolic execution of e under p.

We give the implementation of symbolic_exe in Fig.11. Similar to sep_auto, we first do normalization and destruction to make sure that there are no existential quantifications in the hypothesis and the goal (lines 4-7). Then sep_get_rv is called to calculate the value of the expression. sep_get_rv is implemented by repetitively applying the rules in Fig. 12, and these rules are proved to be sound according to the semantics of assertions. The meanings of these rules are straightforward and we omit the explanation here.

5.4 Linked-list Reversal Example Using hoare_forward

We could use one-line proof script "**repeat** hoare_forward; sep_pure" to prove the in-place linked-list reversal program (shown in Fig. 13).

The pre-condition of the program is presented at line 1, and the post-condition is given at line 8. The pre-condition specifies a single linked list pointed by a pointer variable x, and the temporal variables y and t are not initialized. The post-condition specifies a linked list pointed by variable x with reversed list of values in it.

We need to do hoare_forward 7 times to complete the proof. The 1st hoare_forward assigns NULL to y and generates the post-condition at line 2. The 2nd hoare_forward first proves that the assertion at line 2 implies the loop invariant I, then tries to unfold I with $\mathsf{x} \neq \mathsf{NULL}$, and finally uses WHILE to handle the while loop and convert "$\wedge$istrue($\mathsf{x} \neq \mathsf{NULL}$)" into "$*\langle a_x \neq 0\rangle$" at line 3. Also a similar conversion is done for the post-condition of the while loop at line 7. The 3rd hoare_forward unfolds $\mathsf{lseg}(a_x, 0, L_x)$ and deals with the assignment statement at line 4. The 4th to 6th hoare_forward handle the next 3 assignment statement and prove that the post-condition of the 3rd statement (line 5) implies the loop invariant I. Then the 7th hoare_forward handles the last assignment statement and proves that the assertion at line 7 implies the assertion at line 8. Finally we use sep_pure to prove the remaining pure subgoals to finish the proof for the whole program.

5.5 Debugging with hoare_forward

Our hoare_forward tactic provides some nice features for debugging proofs. If hoare_forward fails with some error message and does nothing over the current Hoare judgement, we know that the resource specified by the pre-condition does not match the resource required by the current statement. We need to adjust the pre-condition or the code in terms of the error message to make the execution of hoare_forward succeeds. If hoare_forward succeeds and generates some pure subgoals which cannot be proved, then we may need to add additional pure assertions in the pre-condition or modify the code to get additional conditions to solve them. We will demonstrate these features using the following example.

$$\frac{}{\vdash \{\mathsf{x} \rightarrowtail \_ * \mathsf{y} \rightarrowtail \_\}\mathsf{y} = (*\mathsf{x}).\mathsf{next}\{\mathsf{x} \rightarrowtail \_ * \mathsf{y} \rightarrowtail \_\}}$$

Our tactic hoare_forward fails to step forward and does nothing because the execution of hoare_unfold over $(*\mathsf{x}).\mathsf{next}$ with the pre-condition fails. The expression $(*\mathsf{x}).\mathsf{next}$ demands $\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * a{+}4 \mapsto \_$, while the pre-condition does not have it. Then we check the pre-condition and add a list segment pointed by x, but we forget to add the same resource in the post condition:

$$\frac{}{\vdash \{\mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_\}\mathsf{y} = (*\mathsf{x}).\mathsf{next}\{\mathsf{x} \rightarrowtail \_ * \mathsf{y} \rightarrowtail \_\}}$$

Then hoare_forward first unfolds the list segment and generates a pure subgoal which requires that the head node of the list segment is not NULL,

$$\frac{H : \sigma \models \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \mathsf{y} \rightarrowtail \_}{a \neq 0}$$

$$I \triangleq \left\{\begin{array}{l}\mathsf{Ex}\ a_x, L_x.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a_x) * \mathsf{lseg}(a_x, 0, L_x) * \\ \mathsf{Ex}\ a_y, L_y.\mathsf{y} \rightarrowtail \mathsf{Vptr}(a_y) * \mathsf{lseg}(a_y, 0, L_y) * \\ \mathsf{t} \rightarrowtail \_ * \langle L_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle\end{array}\right\}$$

1 $\{\ \mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_ * \mathsf{t} \rightarrowtail \_\ \}$

$\mathsf{y} = \mathsf{NULL};$

2 $\{\ \mathsf{y} \rightarrowtail \mathsf{NULL} * \mathsf{list}(\mathsf{x}, L) * \mathsf{t} \rightarrowtail \_\ \}$

while $[I]$ $(\mathsf{x} \neq \mathsf{NULL})\{$

$$3\quad \left\{\begin{array}{l}\mathsf{Ex}\ a_x, a_y, L_x, L_y.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a_x) * \\ \mathsf{lseg}(a_x, 0, L_x) * \mathsf{y} \rightarrowtail \mathsf{Vptr}(a_y) * \\ \mathsf{lseg}(a_y, 0, L_y) * \mathsf{t} \rightarrowtail \_ * \\ \langle L_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle * \langle a_x \neq 0 \rangle\}\end{array}\right\}$$

$\quad \mathsf{t} = (*\mathsf{x}).\mathsf{next};$

$$4\quad \left\{\begin{array}{l}\mathsf{Ex}\ a_x, a_y, L_x, L_y, v, L'_x, a.\mathsf{t} \rightarrowtail \mathsf{Vptr}(a) * \\ \langle L_x = v :: L'_x \rangle * \mathsf{x} \rightarrowtail \mathsf{Vptr}(a_x) * a_x \mapsto v * \\ a_x{+}4 \mapsto \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L'_x) * \\ \mathsf{y} \rightarrowtail \mathsf{Vptr}(a_y) * \mathsf{lseg}(a_y, 0, L_y) * \\ \langle v :: L'_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle * \langle a_x \neq 0 \rangle\end{array}\right\}$$

$\quad (*\mathsf{x}).\mathsf{next} = \mathsf{y};$
$\quad \mathsf{y} = \mathsf{x};$
$\quad \mathsf{x} = \mathsf{t};$

$$5\quad \left\{\begin{array}{l}\mathsf{Ex}\ a_x, a_y, L_x, L_y, v, L'_x, a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \\ \mathsf{y} \rightarrowtail \mathsf{Vptr}(a_x) * a_x{+}4 \mapsto \mathsf{Vptr}(a_y) * \\ \mathsf{t} \rightarrowtail \mathsf{Vptr}(a) * \langle L_x = v :: L'_x \rangle * a_x \mapsto v * \\ \mathsf{lseg}(a, 0, L'_x) * \mathsf{lseg}(a_y, 0, L_y) * \\ \langle v :: L'_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle * \langle a_x \neq 0 \rangle\end{array}\right\}$$

$\}$

$$6\quad \left\{\begin{array}{l}\mathsf{Ex}\ a_x, a_y, L_x, L_y.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a_x) * \\ \mathsf{lseg}(a_x, 0, L_x) * \mathsf{y} \rightarrowtail \mathsf{Vptr}(a_y) * \\ \mathsf{lseg}(a_y, 0, L_y) * \mathsf{t} \rightarrowtail \_ * \\ \langle L_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle * \langle a_x = 0 \rangle\end{array}\right\}$$

$\mathsf{x} = \mathsf{y};$

$$7\quad \left\{\begin{array}{l}\mathsf{Ex}\ a_x, a_y, L_x, L_y.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a_y) * \\ \mathsf{lseg}(a_x, 0, L_x) * \mathsf{y} \rightarrowtail \mathsf{Vptr}(a_y) * \\ \mathsf{lseg}(a_y, 0, L_y) * \mathsf{t} \rightarrowtail \_ * \\ \langle L_x \mathbin{++} \mathsf{rev}(L_y) = L \rangle * \langle a_x = 0 \rangle\end{array}\right\}$$

8 $\{\ \mathsf{list}(\mathsf{x}, \mathsf{rev}(L)) * \mathsf{y} \rightarrowtail \_ * \mathsf{t} \rightarrowtail \_\ \}$

Figure 13. Prove In-Place List Reversal

and we have the Hoare judgement as follows:

$$\frac{}{\vdash \left\{\begin{array}{l}\mathsf{Ex}\ a, v, L', a'. \\ \langle L = v :: L' \rangle * a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \\ \mathsf{lseg}(a', 0, L') * \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{y} \rightarrowtail \_\end{array}\right\} \begin{array}{l} \\ \mathsf{y} = (*\mathsf{x}).\mathsf{next} \\ \{\mathsf{x} \rightarrowtail \_ * \mathsf{y} \rightarrowtail \_\}\end{array}}$$

Then it steps forward with the ASSIGN rule and instantiates the post-condition of the assignment statement with the assertion in H. Besides the pure subgoal $a \neq 0$ generated before, another goal is to prove that the assertion in H implies the given post-condition:

$$\frac{\begin{array}{l}H : \sigma \models \mathsf{Ex}\ a, v, L', a'.\mathsf{y} \rightarrowtail \mathsf{Vptr}(a') * \langle L = v :: L' \rangle * a \mapsto v * \\ \qquad a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L') * \mathsf{x} \rightarrowtail \mathsf{Vptr}(a)\end{array}}{\sigma \models \mathsf{x} \rightarrowtail \_ * \mathsf{y} \rightarrowtail \_}$$

Finally hoare_forward will call sep_auto to solve this goal. After eliminating all the matched spatial assertions in the hypothesis and the goal, we get an unprovable goal as below:

$$\frac{\begin{array}{l}H_1 : L = v :: L' \\ H : \sigma \models a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L')\end{array}}{\sigma \models \mathsf{emp}}$$

Then we would know that there should be a list segment $\mathsf{lseg}(a, 0, L)$ in the post-condition, and we modify the post-condition as below:

$$\frac{}{\vdash \{\mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_\}\mathsf{y} = (*\mathsf{x}).\mathsf{next}\{\mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_\}}$$

Now hoare_forward will successfully step forward the Hoare judgement and prove the generated post-condition implies the one that we give. However, when hoare_unfold checks the resource of $(*\mathsf{x}).\mathsf{next}$, it generates a pure subgoal $a \neq 0$ which cannot be proved under the current proof context. That is because the pre-condition does not provide proper knowledge for a and L. To fix this bug, we may simply add a pure assertion in the pre-condition or modify the code. For example, we can change the pre-condition into $\mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_ * \langle L \neq \mathsf{nil} \rangle$, then we have an additional condition $H1$, which makes the pure subgoal provable :

$$\frac{H1 : L \neq \mathsf{nil} \qquad H : \sigma \models \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \mathsf{y} \rightarrowtail \_}{a \neq 0}$$

Also we could modify the code to satisfy the specification. Figure 14 presents the fixed code that satisfies the given specifications. We add a conditional statement for reading $(*\mathsf{x}).\mathsf{next}$. Then hoare_forward solves the whole goal automatically since it is trivial to prove the assertion at line 2 implies $a \neq 0$.

1 $\{\ \mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_\ \}$

if $(\ \mathsf{x} \neq \mathsf{NULL}\ )\ \{$

2 $\{\ \mathsf{Ex}\ a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle a \neq 0 \rangle\ \}$

$\quad \mathsf{y} = (*\mathsf{x}).\mathsf{next};$

$$3\quad \left\{\begin{array}{l}\mathsf{Ex}\ a, v, L', a'.\mathsf{y} \rightarrowtail \mathsf{Vptr}(a') * \langle L = v :: L' \rangle * \mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \\ a \mapsto v * a{+}4 \mapsto \mathsf{Vptr}(a') * \mathsf{lseg}(a', 0, L') * \langle a \neq 0 \rangle\end{array}\right\}$$

$\}$ **else** $\{$

4 $\{\ \mathsf{Ex}\ a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle a = 0 \rangle\ \}$

$\quad$ **skip**;

5 $\{\ \mathsf{Ex}\ a.\mathsf{x} \rightarrowtail \mathsf{Vptr}(a) * \mathsf{lseg}(a, 0, L) * \langle a = 0 \rangle\ \}$

$\}$

6 $\{\ \mathsf{list}(\mathsf{x}, L) * \mathsf{y} \rightarrowtail \_\ \}$

Figure 14. Fix the Proofs by Modifying the Code

6. Implementation and Evaluation

Our tactics are implemented entirely in the Coq8.4 tactic language Ltac. The tool suite consists of 18000 lines of definitions and lemmas, among which there are about 3000 lines for implementing the tactics, while the others are for the definition of the subset of C, the memory model, the separation logic assertions, the inference rules of the program logic and associated lemmas. We have used the tactics described in this paper to verify some simple programs, as shown in Table 1. Except for dlist_traverse that traverses the double-linked list, all the other programs can be proved with the tactic "**repeat** hoare_forward; sep_pure", and they only use one-line proof and 4 words of proof scripts. We carry out our implementation using a machine with 8-core 2.5GHz cpu and 8G mem-

Programs	Code Lines	Proof Lines	Proof Words
swap	3	1	4
swap_struct	9	1	4
list_reversal	8	1	4
list_traverse	4	1	4
list_enqueue	10	1	4
list_append	10	1	4
list_stack_push	2	1	4
list_stack_pop	6	1	4
dlist_traverse	4	70	162

Table 1. Some Examples Proved by Our Tactics

ory, and it takes less than 30 seconds to finish the proofs for these examples.

Since our hoare_unfold does not support automated unfolding for double-linked lists, we cannot prove the dlist_traverse example with one-line proof script. We need to manually do unfolding before running hoare_forward, and to prove manually some generated side-conditions as well. Although it takes about 70 lines to complete the proofs for the 4-line code, comparing to the naive Coq proofs (about 200 lines of proofs for each line of code) in our previous work [10, 11], this is much better and we already benefit a lot from our tactics. For the future work, we could extend our hoare_unfold tactic to unfold double-linked lists automatically for better automation support.

Table 2 compares our tactics with some existing Coq tactics in related work in proving the simple in-place list reversal function. It gives the evidence to show that our tactics are “mostly-automated” as Bedrock [8] for verifying programs manipulating singly-linked lists, and outperform the other three. Detailed comparison with these implementations is given in Sec. 7 below.

Tool Suites	Proof Lines	Proof Words
VST [2]	200	795
SLTK [16]	68	400
Charge [6]	25	105
Bedrock [8]	1	3
Our Tactics	1	4

Table 2. Comparison of Proof Scripts for In-place List Reversal

7. Related Work

Bedrock [8] is a framework which uses computational higher-order separation logic and supports mostly-automated proofs about low-level programs. Unlike our tactics, Bedrock requires the user to annotate the source code with hints for lemma applications (like list rolling and unrolling), and a tactic named “vcgen” is developed to generate verification conditions which can be solved by another tactic like our sep_auto. We try to use Bedrock to prove some examples, and our experience is that when the generated verification conditions cannot be solved they are too complicated for users to figure out what the problems are. It takes us lots of effort to adjust the specifications and code to achieve automated proofs. Our hoare_forward tactic supports step-by-step forward reasoning, and users could get useful information for debugging the specifications and the code from the unproven goals as we have explained before. Our tactics are also “mostly-automated”, especially for programs manipulating singly-linked lists. For example, as shown in Table 2, a useful comparison comes from a simple in-place linked list reversal function implemented in both systems. None of our tactics mentions variable or hypothesis names that are bound within the proof as the Bedrock proof does. We do not consider higher-order separation logic and our tactics only deal with the singly-linked lists for now, also our tactics do not support data structures such as double-linked lists and trees, which we leave as future work.

McCreight’s Coq tactics [16] for separation logic address similar concerns to the tactics presented in this paper. We borrow some ideas from his work to implement sep_lift and sep_simpl, which are called by sep_auto and hoare_forward. Proofs using McCreight-s tactics still involve a significant number of manual proof steps, applying operations such as explicit rearrangement of separating conjunctions using associativity and commutativity. Since his implementation also uses a vc-gen to verify programs it has the similar weakness on debugging proofs as Bedrock. Our sep_auto and hoare_forward hide these operations and have better support for debugging proofs. The number of atomic tactic calls included in our proofs for the simple in-place linked list reversal function are much less than that in his proofs.

Appel’s unpublished note [2] and book [4] describe tactics implemented in Coq for manual verification in a proof assistant using separation logic. It provides a similar tactic named Forward to apply inference rules to move one step according to the syntax of current statements. His Forward tactic simply applies inference rules and generate some side conditions, which have to be proved manually. It does not consider automatically unfold list segments for automated symbolic execution of expressions, thus users have to do lots of interactive proofs to prove programs manipulating linked-lists. Our hoare_forward tactic provides much better automated verification support.

Smallfoot [7] is an automated tool for verifying lightweight separation logic specifications of programs. It achieves fully automated reasoning using a limited separation logic assertions. This approach has been used as the basis for certified separation logic decisions procedures in Coq [3] and HOL [18]. Our tactics do not pursue fully automated verification, but try to provide as much automated verification support as possible for verifying C programs using general and expressive separation logic specifications. Our ultimate goal is to provide proof automation for large scale interactive proof development such as OS kernel verification, and the expressiveness of the assertion language is crucial for this job. Smallfoot cannot be applied doing this since its lightweight separation logic specifications are not expressive enough.

Appel [3] develops a variation of smallfoot shape analyser in Coq and verifies its correctness. Like smallfoot, it cannot be applied for general purpose program verification either due to the limited expressiveness of the assertions. On the other hand, our efforts to support automated reasoning about programs manipulating singly-linked lists are like developing proof-generating shape analysis algorithms. The key difference is that we apply it with other tactics together for general interactive verification with expressive assertions.

Charge [6] provides a set of tactics for working with a higher-order separation logic for a subset of Java in Coq. Their sl_apply tactics does the same thing as our cancel_same tactic, which removes the identical spatial assertions in the hypothesis and the goal. We provide a more powerful tactic sep_cancel, which is able to deal with spatial assertions of list segments when the goal is provable but there do not exist any identical components. Also, their tactic forward-tactic for proving Hoare triples does not deal with singly-linked lists as ours. As a result they need to use 25 lines and 105 words of proof scripts to prove the standard in-place list reversal algorithm, while we only need one line and 4 words.

There is also other work on automated verification of system software. Hawblitzel and collaborators have done automated verification of garbage collectors [12] and the core of an operating system kernel [19] with Boogie [5]. There is no explicit proof terms for their verification. Comparing to these work, our verification is carried out in Coq, which provides proof terms for the verified programs.

8. Conclusion and Future Work

We have implemented a set of practical tactics for verifying C programs in Coq, including sep_auto, sep_normal, sep_lift, sep_split, sep_cancel and cancel_same for automatically proving separation logic assertions and hoare_forward for automatic verification condition generation. In particular, we develop special tactics hoare_unfold for verifying programs manipulating linked lists. Using our tactics we are able to verify several C programs with one-line proof script. The key feature of our tactics is that, if the tactics fail, they allow users to easily locate problems causing the failure by looking into the remaining subgoals, which greatly improves the usability when human interaction is necessary.

The purpose of our tactics is for verifying a variant of μC/OS-II. For the future work, we will do the following things to apply our tactics in OS kernel verification. We will extend our tactics to support more data structures, such as double-linked lists and trees. We will extend our sep_pure to support automatically proving the arithmetic properties of *Int32*. Also we will adapt our tactics to support relational reasoning for proving linearizability of APIs provided by μC/OS-II that has fine-grained concurrency in the kernel.

Acknowledgments

We thank anonymous referees for their suggestions and comments. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant Nos. 61103023, 61229201, 61379039 and 91318301, the National Hi-Tech Research and Development Program of China (Grant No. 2012AA010901) and the Fundamental Research Funds for the Central Universities (Grant No. WK0110000031). Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] The coq development team: The coq proof assistant. http://coq.inria.fr.

[2] A. W. Appel. Tactics for separation logic, 2006. http://www.cs.princeton.edu/~appel/papers/septacs.pdf.

[3] A. W. Appel. Verismall: Verified smallfoot shape analysis. In *Proceedings of Int'l Conf. on Certified Programs and Proofs (CPP'11)*, pages 231–246, 2011.

[4] A. W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*, pages 364–387, 2006.

[6] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - a framework for higher-order separation logic in coq. In *Proceedings of Interactive Theorem Proving (ITP'12)*, pages 315–331, 2012.

[7] J. Berdine, C. Calcagno, and P. W.O'Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects (FMCO'05)*, pages 115–137, 2005.

[8] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI'11)*, pages 234–245, 2011.

[9] D. Delahaye. A tactic language for the system coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAP'00)*, pages 85–95, 2000.

[10] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 170–182, 2008.

[11] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 401–414, 2006.

[12] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'09)*, 2009.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220, 2009.

[14] J. J. Labrosse. *Microc/OS-II*. R & D Books, 2nd edition, 1998.

[15] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, pages 42–54, 2006.

[16] A. McCreight. Practical tactics for separation logic. In *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs'09)*, pages 343–358, 2009.

[17] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002.

[18] T. Tuerk. A separation logic framework in HOL. In *Proceedings of Theorem Proving in Higher Order Logics: Emerging Trends*, pages 116–122, 2008.

[19] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 99–110, 2010.