Separation-Logic-Based Program Verification in Coq

Qinxiang Cao

A Dissertation
Presented to the Faculty of Princeton University
in Candidacy for the Degree of Doctor of Philosophy

Recommended for Acceptance by the Department of Computer Science
Adviser: Andrew W. Appel

September 2018
Abstract

In our interconnected world, software bugs can seriously compromise our safety and security. To provide adequate safety or protection, security-critical kernels (of large systems) must be functionally correct.

To ensure functional correctness of C programs, we can use Hoare logic and its extensions such as separation logic. But such correctness proofs are large and complex enough that we cannot trust them unless they are machine-checked.

This dissertation shows how we can construct machine-checked proofs of program correctness using separation logic. I answer three questions: How shall we specify our programs? How shall we prove our programs correct with respect to their specifications? How shall we mechanize such correctness proofs?

I present practical techniques for separation-logic-based program verification and demonstrate them on several examples. I introduce VST-Floyd, a tool for users to build formal C program correctness proofs in Coq using separation logic. VST-Floyd is built based on Verifiable C, a proved sound separation logic; I show how to reformulate its rules to make them practical for use in verification.
## Contents

Abstract ................................................................. iii

1 Introduction .......................................................... 1

1.1 Background: Hoare logics and separation logics ............... 1

1.2 Background: automated verification and interactive verification ... 4

1.3 Background: verification in Coq .................................. 5

1.4 Background: Verified Software Toolchain ....................... 8

1.5 Organization of this thesis ........................................ 9

2 Preliminaries .......................................................... 10

2.1 C programs .......................................................... 10

2.2 Assertions .......................................................... 13

2.3 Hoare triples ....................................................... 15

2.4 Judgments ........................................................... 17

2.5 Coq proof goals as judgments ................................... 18

2.6 Proof rules .......................................................... 20

2.6.1 Generic proof rules for entailments ......................... 20

2.6.2 Domain-specific proof rules for entailments ............. 23

2.6.3 Proof rules for Hoare triples ............................... 25

2.7 Coq formalization of proof rules ................................. 27

2.8 Semantics and soundness .......................................... 28
I Magic wand as frame

3 Previous work using magic wand

4 Discussion: the limited proof power
   4.1 A simple semantics
   4.2 Distinguish sound rules from unsound rules
   4.3 Summary

5 Wand frame proof rules
   5.1 Wand-frame rules
   5.2 WandQ-frame rules
   5.3 WandP-frame rules

6 Case study: linked list append
   6.1 Wand frame proof
   6.2 WandQ frame proof
   6.3 Another wandQ frame proof
   6.4 Comparison with recursive predicates

7 Case study: binary search tree insert
   7.1 Separation logic specification of implementation correctness
   7.2 Magic wand for partial trees
   7.3 Implementation correctness proof
   7.4 Other BST operations

8 Improving ramification
12.2.1 Derived proof rules ........................................ 122
12.2.2 Computing syntactic path .................................. 124
12.2.3 Computing load/store addresses ............................ 126
12.2.4 Improving path representation via user’s hint............ 131
12.2.5 Searching in SEP clauses and generating postconditions ................................. 136
12.2.6 Summary ..................................................... 138

12.3 Function calls .................................................. 139

13 Forward verification .............................................. 143
13.1 Forward through set, load and store commands ............ 143
13.1.1 Implementation............................................. 145
13.1.2 Reassociating sequences .................................... 147
13.2 Forward through function calls ............................... 148
13.3 Forward proof through if commands .......................... 149
13.4 Forward proof through loops ................................ 154
13.5 Weakening preconditions ..................................... 155
13.5.1 Leveraging Coq’s built-in tactics ........................... 156
13.5.2 Gather SEP and Replace SEP ................................ 157
13.5.3 Flattening the SEP clauses ................................ 159
13.6 Extracting existentials and propositions ..................... 160
13.6.1 Extracting PROP clauses ................................ 161
13.6.2 Extracting existentials .................................... 163
13.6.3 Common scenario: Intros after unfolding user-defined predicates .......................... 165
13.6.4 Discussion: are extraction rules necessary? ............... 166
13.7 Introducing assumptions that are derivable from preconditions ..................................... 168
13.8 Summary ..................................................... 169
Chapter 1

Introduction

Getting rid of program bugs is a battle from the first day people developed software. In practice, software engineers try to avoid most program bugs, especially catastrophic ones, by testing and code review.

However, these approaches cannot cover all possible cases. In the past several decades, the software industry has developed dramatically and software has become huge. Program bugs that hide in corners are almost inevitable in million-line software. Some software components are especially safety-critical. They must be fully bug-free.

In this thesis, I present new techniques in program verification—proving that a program is correct under all possible cases. Specifically, this thesis describes new techniques in separation-logic-based interactive verification.

1.1 Background: Hoare logics and separation logics

Hoare logic \cite{29} is a traditional way to specify and prove program correctness.

Hoare triples use assertions to describe programs’ behavior: a precondition $P$, a program $c$ and a postcondition $Q$ form a Hoare triple $\{P\} c \{Q\}$. Basically, a triple
\{P\} \ c \ \{Q\} \text{ means}^{1}

For any initial state $s$ and ending state $t$, if $s$ satisfies $P$ and running $c$ from $s$ may terminate in state $t$, then $t$ satisfies $Q$.

Hoare logics are compositional, i.e. a Hoare triple of a program can be derived from the Hoare triples of its components. For example, here is the rule for sequential composition:

$$
\text{hoare-seq} \quad \frac{\{P\} \ c_1 \ \{Q\} \quad \{Q\} \ c_2 \ \{R\}}{\{P\} \ c_1; c_2 \ \{R\}}
$$

Usually, the assertion language of a Hoare logic at least contains propositional connectives: conjunction, implication etc. Separation logic extends Hoare logic’s assertion language with two extra connectives—separating conjunction ($P \ast Q$, or “star”) and separating implication ($P \rightarrow Q$, or “magic wand”).

The separating conjunction $P \ast Q$ in assertions represents the existence of two disjoint portions of the state, one that satisfies $P$ and one that satisfies $Q$. Specifically,

$$
m \models P \ast Q \iff \text{ there exist } m_1 \text{ and } m_2 \text{ s.t. } m = m_1 \oplus m_2, m_1 \models P \text{ and } m_2 \models Q.
$$

Here, $m_1 \oplus m_2$ represents the disjoint union of two pieces of state/memory.

Separating conjunction is commutative and associative.

$$
\text{sepcon-comm} \quad \frac{P \ast Q}{Q \ast P} \quad \text{sepcon-assoc} \quad \frac{P \ast (Q \ast R)}{(P \ast Q) \ast R}
$$

Moreover, separating conjunction concisely expresses address (anti)aliasing. For example, if “$p \mapsto v$” is the assertion that data $v$ is stored at address $p$, then $p \mapsto \ldots$

---

1Hoare triples’ interpretation may differ from different separation logics. In some Hoare logic, a Hoare triple $\{P\} \ c \ \{Q\}$ ensures that the program $c$ must terminate if starting executing it from a state satisfying $P$. In some Hoare logic, $\{P\} \ c \ \{Q\}$ ensures that executing $c$ from a state satisfying $P$ will never crash. In this thesis, Hoare triples does not ensure programs’ termination but ensures the crash-free property.
\( v \mapsto q \mapsto u \) says \( v \) is stored at address \( p \), \( u \) is stored at address \( q \), and \( p \neq q \). This is not limited to assertions with only two conjuncts. The assertion \( p_1 \mapsto v_1 \mapsto p_2 \mapsto v_2 \mapsto \ldots \mapsto p_n \mapsto v_n \) implies the fact that \( p_1, p_2, \ldots, p_n \) are different from each other.

Separating implication is a right adjoint of separating conjunction:

\[ m \models P \rightarrow Q \text{ iff. for any } m_1 \text{ and } m_2, \text{ if } m \oplus m_1 = m_2 \text{ and } m_1 \models P \text{ then } m_2 \models Q. \]

\[ \frac{P \vdash Q \rightarrow R}{P \mapsto Q \vdash R} \quad \frac{P \mapsto Q \vdash R}{P \vdash Q \rightarrow R} \]

Separation logic also adds a constant predicate \( \text{emp} \). It is only satisfied on \textit{empty} states. It is a unit of separating conjunction:

\[ \frac{}{P \mapsto \text{emp} \not\models P} \]

Besides basic Hoare logic proof rules like \textsc{hoare-seq}, separation Hoare logic enables one to verify a Hoare triple locally but use it globally, using the \textit{frame rule}:

\[ \frac{}{\{P\} \ c \ \{Q\} \quad \{P \mapsto F\} \ c \ \{Q \mapsto F\}} \]

(if no program variables which freely occurs in \( F \) are modified by \( c \).)

In different articles, “separation logic” can mean either (1) a proof theory of assertion entailments with \( \ast, \ast \) and \( \text{emp} \) or (2) a Hoare logic with \( \ast, \ast \) and \( \text{emp} \) in its assertion language. In the remainder of this thesis, I will use “separation logic” to denote those proof theories of assertion entailments and use “separation Hoare logic” to denote those extended Hoare logics.
1.2 Background: automated verification and interactive verification

A program correctness proof can ensure the corresponding program will be fully bug-free. But this is not a final solution. The verification of a long program can only be an even longer proof. A long proof is not more trustworthy than a long program.

In the past several decades, tools and techniques have been developed for verifying programs’ correctness. They provide trustworthy verification in two different approaches: automated verification [23, 40, 33, 47, 20, 24] and interactive verification [9, 27, 38].

Using automated verification tools, computers take the main responsibility for proving programs’ correctness. Specifically, humans write programs, humans write program specifications and computers prove that the programs obey their corresponding specifications.

Using interactive verification tools, humans are responsible for building correctness proofs. In other words, humans write programs and specifications, humans prove that the programs obey their corresponding specifications and computers check whether every step in these proofs follows basic logical proof rules.

In both cases, after a program gets verified, this program’s correctness does NOT depend on the correctness proof itself, because it has been machine-checked. It only depends on the manually written program specification and the verification tool that proves the correctness (in the automated case) or that checks a human-written proof. In other words, users of these verification tools need only be very careful and make sure that their specifications correctly express the required properties—if they trust their verification tools.

Comparing these two different approaches, the automated approach needs significant less human effort in the verification process. With nearly no human assistance, a
typical modern automated verification tool can verify a thousand-line-code program is arithmetic-overflow-free with 10 seconds (under some practical restrictions). Some advanced tools like F-Soft [31, 32] can even analyze one million lines of code efficiently. In contrast, latest interactive verification tools still require programmers or researchers to write long formal proofs, which can be 5 times or 10 times longer than the program itself.

But because of the undecidability theorem [21], there cannot be a program which is able to check all correctness properties. Thus, automated tools can only verify a limited set of them. For example, generic functional correctness is beyond the scope of automated verification.

Due to this, automated verification is usually used when human resources are not sufficient and the software is less correctness-sensitive. For example, software companies use these tools to catch bugs in legacy code which has been running for years [6].

In contrast, some software is more correctness-sensitive. One example is compilers. We should ensure that compilers always generate correct target code. Another example is those programs which control safety-critical equipments or expensive facilities. If the operating system of a self-driving car is buggy, it can cost lives. In these situations, it is not enough to verify that a compiler or an auto-driver, as a program, does not crash—they must work correctly.

1.3 Background: verification in Coq

Coq [7] is a tool for interactively building proofs. The following are the major components of Coq:

1. A formal language, Gallina. Specifically, the syntax of Gallina is the calculus of inductive constructions (CiC) which allows us to define all kinds of math-
Mathematical concepts in Coq. Gallina (CiC) terms are dependently typed. That is, every legal Gallina term has a type, which is also a Gallina term. If a Gallina term represents a type, then the type of that term itself is “Type” \( \text{Type} \). Coq treats propositions (theorems, lemmas, etc) and their proofs as types and terms according to the Curry-Howard isomorphism.

2. The kernel, a syntactic checker which checks whether a Gallina term \( t \) is a legal term of type \( T \), which is another Gallina term.

3. A programmable tactic language, Ltac. Users of Coq do not usually build proof terms directly in Gallina. Coq offers Ltac, instructions to build proof terms. We call these Ltac programs proof scripts.

4. An interpreter. It constructs proof terms according to the instructions in proof scripts.

In order to formally prove a theorem in Coq, we usually follow the steps below:

1. State that theorem in Coq. For example, the following is a Coq theorem statement.

   \[
   \text{Theorem no\_greatest\_nat:} \\
   \text{forall n: nat, exists m: nat, n < m.}
   \]

2. Write a proof script \( p \) in Coq interactively with the help of Coq’s interpreter. For example, a proof script for the theorem above can be the following one:

   \[
   \text{intros. exists (n+1). omega.}
   \]

Specifically, CoqIDE (or any other IDE for Coq) demonstrates the remaining proof goals in a window beside the editing script and helps users figure out what to do next.

\(^2\)Strictly speaking, Type is not a Gallina term. It represents a term of form Type@\( n \) where \( n \) is a universe. Details can be found in *Universe Polymorphism in Coq*. 

\[50\]
Figure 1.1: Coq Proof Scripts and Proof Goals

In every proof goal, the items above the line are assumptions and the proposition below the line is the conclusion. Here, intros pulls a universally quantified variable into assumptions. The tactic exists instantiates an existentially quantified variable and omega solves inequalities in integer linear arithmetic.

3. Let Coq’s interpreter build a proof term $t$ according to the proof script $p$.

4. Suppose the theorem statement is $T$. Let the Coq kernel check whether $t$ is actually a legal proof term of $T$. This step is done when users use the Qed command to finish a proof script. After that, this proved theorem can be used in further proofs.

Coq is intentionally designed like this such that the trusted base of every Coq-formalized theorem is very small. (Something is in the trusted base if we have to trust its correctness. Something is outside the trusted base if its correctness is checked or
certified by other components.) The proof script \( p \) and Coq’s interpreter are not in the trusted base since they are only used to generate the proof term \( t \). The proof term \( t \) is also not in the trusted base because Coq’s kernel checks it. In the end, the trusted base only contains the statement \( T \) and the kernel.

Here, it is very important that even the interpreter is not in the trusted base. Although it is smaller than those long, complex proofs (either \( p \) or \( t \)), it is nontrivial compared to the kernel. In the past decade, new versions of Coq have been released to support new features in the tactic languages. Even some simple automated solvers have been built in as Coq standard tactics. In contrast, the Gallina language and kernel are relatively stable and are kept simple and clean.

### 1.4 Background: Verified Software Toolchain

Gallina is a very rich language. It allows Coq’s users to define all kinds of mathematical concepts and state all kinds of properties. As a result, many different theoretical results can be formalized in one single ecosystem. For example, the soundness proof of Hoare logics and Hoare logic proofs for program correctness can both be formalized in Coq.

Verified Software Toolchain (VST) [2] is a research project that formalizes a separation Hoare logic for C language, proves it sound with respect to CompCert C semantics [43], and enables users to use this logic to prove their C programs correct. VST connects these formalized components together in Coq. Figure 1.2 shows VST’s main structure.

Specifically, Verifiable C (a component of VST) proves a separation Hoare logic sound with respect to the operational semantics of Clight (an AST of C defined in Coq by CompCert) and its memory model. Users can use this separation Hoare logic proof rules sound. These rules of course form a separation Hoare logic’s proof theory. But Verifiable C does not treat them as
logic to prove their C program correct. Furthermore, that Hoare logic soundness proof is connected to a compiler correctness proof. That is, composing these three components together means that every proved-correct C program will be compiled to an assembly program with expected behavior.

1.5 Organization of this thesis

This thesis contributes to two parts in this toolchain. Part I of this thesis proposes a new separation Hoare logic proof strategy called magic-wand-as-frame. Part II introduces VST-Floyd which helps users build Hoare logic proofs more conveniently. It extends the separation Hoare logic provided by Verifiable C and provides tactics to handle a lot of C semantic subtleties.

These two parts correspond to two double-lined circles in Figure 1.2.

Some major results in thesis have already been published in JAR 2018 [17].
Chapter 2

Preliminaries

2.1 C programs

This thesis uses C programs as examples of software verification. VST-Floyd (introduced in Part [1]) is specifically designed for C program verification. The proof technique demonstrated in Part [1] should work for any separation Hoare logic for any imperative language. I always use C programs so that they can be reused across chapters. But the results and the techniques are not only for C-specific verification.

I list some C program expressions and commands here:

C Program variables:

\[ x, y, i, \text{sum} \ldots \]

C expressions:

\[ x + y, a[10], p.\text{next} \ldots \]

C Commands:

\[ x = x + 1, \text{if}(p == \text{NULL}) c_1 \text{ else } c_2, \ldots \]
Later in this thesis, I use the letter $e$ to range over C program expressions and $c$ to range over C program commands.

Following the formalization approach of Clight, this thesis distinguishes addressable variables and nonaddressable variables. In the C language, one can take the address of variable $x$ by the syntax `&x`. Any scalar (int, float, pointer) local variable whose address is never taken (a property easily determined statically) is called nonaddressable. All other variables—globals, aggregates (`struct`, `union`, arrays), and those whose address is taken—are addressable. Addressable variables are stored in memory, while nonaddressable variables are typically kept in machine registers.

We pay attention to the difference because, in a program logic, nonaddressable variables can often be reasoned about by substitution, instead of the heavyweight mechanism of separation.

**Primary r-value expressions**, **primary l-value expressions** and four kinds of primary assignment commands play an important role in the proof rules in this thesis. A **primary r-value expression** does not contain any memory dereferences or function calls. A **primary l-value expression** is an expression that refers to an address in memory and such that the computation of the address does not involve any memory dereferences or function calls. For example, if $x$ and $y$ are nonaddressable variables of integer type and $a$ is a variable of integer array type, then $x+y$, $x\times x$, $x\leq y$, $a+x$, $&(a[x + 1])$ are primary r-value expressions; $a[0]$, $a[x]$, $a[x+1]$, $*(a+x)$ are primary l-value expressions; and $a[x]+a[y]$, $a[x]\times 2$, $a[a[x]]$ are not primary expressions.

All primary assignment commands are:

1. Set command: the left side is a nonaddressable variable and the right side is a primary r-value expression, e.g. $x = y + 2$;

2. Load command: the left side is a nonaddressable variable and the right side is a primary l-value expression, e.g. $x = a[y + 2]$;
3. Store command: the left side is a primary l-value expression and right side is a primary r-value expression, e.g. \( a[x] = y + 2 \).

4. Function call: the left side (if present) is a nonaddressable variable and the right side is a function call, e.g. \( x = f(y + 2) \).

When a C language assignment command is nonprimary, we can always split it into primary ones by inserting extra assignments to temporary variables. For example, \( a[x] = a[y] \) will be decomposed into: \( t = a[y]; a[x] = t \). Here \( t \) is a new temporary variable of integer type.

In the programming language literature, authors often prefer to use while loops as the unification of different loop commands. In other words, “\( \text{for } (c_0; b; c_i) \ c \)” would be defined as “\( c_0; \text{while } (b) \ {c; c_i} \)”. But that approach turns out to be problematic for formalizing C. C has continue commands! In “\( \text{for } (c_0; b; c_i) \ c \)”, after reaching a continue in \( c \), the next command to execute is \( c_i \). In “\( c_0; \text{while } (b) \ {c; c_i} \)”, after reaching a continue in \( c \), \( c_i \) will be skipped as well and the next command is to test the loop condition \( b \) directly. Therefore, while cannot be used to implement for.

This thesis follows Clight’s formalization by unifying different loops in C into the form \( \text{loop}(c_i) \ c \). In this general loop command, \( c \) is the loop body and \( c_i \) is the increment command. Specifically, “\( \text{while } (b) \ c \)” is defined as

\[
\text{loop}(); \{ \text{if } (b) /*\text{skip*/}; \text{else break}; c \}
\]

and “\( \text{for } (c_0; b; c_i) \ c \)” is defined as

\[
c_0; \text{loop}(c_i) \ {\text{if } (b) /*\text{skip*/}; \text{else break}; c \}.
\]
2.2 Assertions

Here I define the assertion language for separation Hoare logic.

Constants:

\[ 0, 1, \text{null, max\_signed, max\_unsigned} \ldots \]

Logical variables:

\[ v ::= x, y, p, q, \ldots \]

Logical types:

\[ \tau ::= \text{val, } \mathbb{Z}, \tau_1 \times \tau_2, \tau_1 \rightarrow \tau_2, \text{list } \tau \ldots \]

Logical expressions:

\[ E ::= \text{constants, } v, [e], [e][\vec{E}/\vec{x}], E_1 + E_2, E_1 - E_2, E_1 \cdot E_2, \]
\[ (E_1, E_2), \text{fst}(E), \text{snd}(E), \]
\[ \text{nil, } \text{cons}(E_1, E_2), \quad E_1 \cdot E_2, \]
\[ \text{match } E_1 \text{ with } \text{nil } \Rightarrow E_2 \mid \text{cons}(v_1, v_2) \Rightarrow E_3 \text{ end} \]
\[ E_1(E_2), \quad \lambda v : \tau. \; E, \quad \text{fix} E, \ldots \]

Nonspatial predicates:

\[ E_1 = E_2, \quad E_1 < E_2, \quad E \text{ is even, } \quad E \text{ is a nonempty list, } \ldots \]

Spatial predicates:

\[ e \downarrow, \quad e[\vec{E}/\vec{x}] \downarrow, \quad E_1 \mapsto E_2, \quad \text{listrep}(E_1, E_2), \quad \text{lseg\_rec}(E_1, E_2, E_3) \ldots \]

Assertions:

\[ P ::= \text{nonspatial predicate, spatial predicate, } P_1 \land P_2, \quad P_1 \lor P_2, \quad P_1 \rightarrow P_2, \quad \bot, \]
\[ \forall v : \tau. \; P, \quad \exists v : \tau. \; P, \quad P_1 * P_2, \quad P_1 \leftrightarrow P_2, \quad \text{emp} \]

At the top level, assertions are composed of predicates with propositional connectives, separating connectives (separating conjunction, separating implication and
emp) and quantifiers. The language is higher-order, i.e. a variable may represent a function and it is allowed to quantify over functions. This thesis will use letters $P, Q$ etc. to represent assertions. As expected, $\neg P$, $P_1 \leftrightarrow P_2$ and $\top$ will be treated as the abbreviations of $P \to \bot$, $(P_1 \to P_2) \land (P_2 \to P_1)$ and $\bot \to \bot$ respectively. Function abstraction $(\lambda v : \tau. E)$ and quantifications $(\forall v : \tau. P, \exists v : \tau. P)$ are typed. But I will usually omit types as long as long they can be inferred from the context.

We can define syntactic substitution for both logical variables and program variables. For any $x$, $E_0$, $E$ and $P$, $E[E_0/x]$ and $P[E_0/x]$ represent the expression and assertion which replace every free occurrence of logical variable $x$ in $E$ and $P$ with $E_0$. For any $\bar{x}_0$, $\bar{E}_0$, $E$ and $P$, $E[\bar{E}_0/\bar{x}_0]$ and $P[\bar{E}_0/\bar{x}_0]$ represent the expression and assertion which apply program variable substitution “[\bar{E}_0/\bar{x}_0]” to all subexpressions of form $[e]$ and $[e][\bar{E}/\bar{x}]$, and predicates with form $e \downarrow$ and $e[\bar{E}/\bar{x}] \downarrow$ inside $E$ and $P$.

The notation $[e]$ means the denotation of a primary r-value expression $e$. In other words, if the evaluation of $e$ needs to load from memory or call C functions, the meaning of $[e]$ is unspecified. Besides, $[e][\bar{E}/\bar{x}]$ is the primary denotation which does not use the actual values of $\bar{x}$ but uses $\bar{E}$ as those variables’ values instead.

Predicates are either nonspatial predicates or spatial predicates. The most import spatial predicate is $E_1 \mapsto E_2$ which says a value $E_2$ is stored at address $E_1$. Another spatial predicate $e \downarrow$ means that $e$ is a primary r-value expression and is safe to compute. (Besides, the predicate $e[\bar{E}/\bar{x}] \downarrow$ says that $e$ is a primary r-value expression and is safe to compute if $\bar{E}$ instead of the actual values of $\bar{x}$ are used.) It may be strange that $e \downarrow$ is spatial. Here is the reason. Suppose $p$ is a nonaddressable variable of type int*. Then whether an expression like $p == \text{null}$ can be evaluated depends on whether $[p]$ is a dangling pointer. If yes, evaluating $p == \text{null}$ is undefined behavior.

---

1 In Clight’s semantics, the Coq type val represents all C values. The value of a C expression is either an integer, a floating point number or an address. A val is either one of these kind of values or Vundef which is used to represent undefined or uninitialized values. Here, if the evaluation of $e$ needs to load from memory or call C functions, $[e]$ is defined as Vundef.
according to C11 standard [1]. Whether \([p]\) is a dangling pointer is a memory-related fact.

The assertion language is flexible in its expression syntax and predicates. It allows all kinds of domain-specific expressions and predicates, for example denotations of program expressions (\([e], e \downarrow\)), arithmetics and comparison, pairs and projection, list (nil and cons \((E_1, E_2)\)) and case analysis (match \ldots \text{with} \ldots), function application and function abstraction, recursive function, etc. Later in this thesis, I will use \([E_1; E_2; \ldots; E_n]\) to represent the abbreviation of cons \((E_1, \text{cons}(E_2, \ldots \text{cons}(E_n, \text{nil}))\)). Also, I overload the dot operator: for list operation, \(E_1 \cdot E_2\) represents concatenation of two lists. For arithmetic, \(E_1 \cdot E_2\) represents multiplication.

There is only one restriction on the domain-specific part of the assertion language: expressions and predicates should be type-checked. For example, \(x + [y] - (0, 1) > 0\) is not a legal predicate because arithmetics cannot be applied on pairs. Also, quantifiers are typed. For example, in the assertion \(\exists f. f([x]) = 0\), \(f\) is quantified over functions from scalar values to scalar values.

### 2.3 Hoare triples

In this thesis, Hoare triples will be used to describe the specification of both C functions and C programs. I use two different syntax to distinguish them.

A C function’s specification is a parameterized Hoare triple with the following form:

\[
\text{WITH } \overrightarrow{x}. \text{PRE}: P_1 \text{ POST}: P_2.
\]

For example, suppose \texttt{swapint} is a C function swapping two integers stored in memory:

\[
\text{void swapint(int } \ast p, \text{ int } \ast q);
\]
Then, its specification can be described as follows:

\[
\text{swapint} : \quad \text{WITH } p, q, x, y.
\]

**PRE:** \([p] = p \land [q] = q \land p \mapsto x \ast q \mapsto y\)

**POST:** \(p \mapsto y \ast q \mapsto x\)

It says, for any value \(p, q, x\) and \(y\), if (1) \(p\) has value \(p\), (2) \(q\) has value \(q\), (3) address \(p\) in the memory stores value \(x\) and (4) another different address \(q\) in the memory stores value \(y\) in the beginning program state of \text{swapint}, then executing \text{swapint} is safe (i.e. will not have run time errors) and address \(p\) and \(q\) will store values \(y\) and \(x\) respectively in the ending states.

A C command’s specification is a Hoare triple with multiple postconditions:

\[
\{P_1\} c \{P_2, [P_3, P_4, P_5]\}
\]

In this Hoare triple, \(P_3\), \(P_4\) and \(P_5\) represent break condition, continue condition and return condition. \(P_1\) and \(P_2\) are pre/postcondition as in the usual notation. In detail, this Hoare triple says: if the beginning state of C command \(c\) satisfies \(P_1\), executing \(c\) is safe and the ending state will satisfy \(P_2, P_3, P_4\) or \(P_5\) if the execution terminates normally, by \text{break}, by \text{continue} or by \text{return}.

Hoare triples denote partial correctness in this thesis. That is, the validity of a Hoare triple does not require a C function/program to terminate.
2.4 Judgments

There are two kinds of judgment in a Hoare logic: assertion entailments and Hoare triples.

Logical contexts: \[ \Sigma ::= \vdash, \nu; \Sigma \]

Assumptions: \[ \Gamma ::= \vdash, \text{P}^{\text{Pure}}; \Gamma \]

C-language contexts: \[ \Delta ::= \vdash, f: \text{WITH}^{\text{xy}}. \text{PRE:} P_1 \text{ POST:} P_2; \Delta \]

Entailments: \[ \Sigma; \Gamma \vdash P \]

Hoare triples: \[ \Sigma; \Gamma; \Delta \vdash \{P_1\} c \{P_2, [P_3, P_4, P_5]\} \]

Also, I will use \( \Sigma; \Gamma; P_1 \vdash P_2 \) and \( \Sigma; \Gamma; P_1 \nvdash P_2 \) as abbreviations of \( \Sigma; \Gamma \vdash P_1 \rightarrow P_2 \) and \( \Sigma; \Gamma \vdash P_1 \leftrightarrow P_2 \) respectively.

In these two kinds of judgments (entailments and Hoare triples), the symbol “\( \vdash \)” is overloaded. Usually, it is clear from the context whether a judgment is an entailment or a triple. In judgments, \( \Sigma \) is a list of logical variables which are allowed to freely occur. \( \Gamma \) represents assumptions, which is a list of \textit{pure} assertions. An assertion is \textit{pure} if and only if it contains no subexpression of form \( e \Downarrow \) or \( [e] \), or any spatial predicate. Suppose \( x \) and \( y \) are two logical variables and \( z \) is a program variable. Then \( "x + y > 0 \land x + y < 1" \) is a pure assertion but neither \( x < [z] \) nor \( x \mapsto y \) is pure. And in Hoare triple judgments, \( \Delta \) is another set of assumptions, a list of C function specifications.

Informally, an entailment judgment \( \Sigma; \Gamma \vdash P \) says, the conjunction of \( \Gamma \) implies \( P \). A Hoare triple judgment \( \Sigma; \Gamma; \Delta \vdash \{P_1\} c \{P_2, [P_3, P_4, P_5]\} \) says, under the assumptions in \( \Gamma \), if every function called by \( c \) satisfies its corresponding specification in \( \Delta \), then the triple \( \{P_1\} c \{P_2, [P_3, P_4, P_5]\} \) is valid.
It seems redundant to have pure assertions as assumptions of entailments. For example, the following entailments do express the same meaning:

\[
\Sigma; (\cdot; P^\text{Pure}) \vdash Q \\
\Sigma; \vdash P^\text{Pure} \rightarrow Q
\]

I define the syntax of judgments like this so that both kinds of judgments can share a single format. Also, in the next section, I will show that this judgment syntax corresponds to the format of Coq proof goals.

Later in this thesis, I will not stick to the letter “\(E\)” for representing expressions. Letters like \(x, y, a, b, p\) and \(q\) may all be used. This might be problematic since it can be ambiguous whether \(x\) is a logical variable or an expression. Sometimes, it is clear because \(x\) has been used as a bound variable by quantifiers. For ambiguous situations, I will explicitly list \(x\) in the context \(\Sigma\) if \(x\) is a variable instead of an expression.

### 2.5 Coq proof goals as judgments

Judgments correspond straightforwardly to Coq proof goals. Logic variables in \(\Sigma\) correspond to the Coq variables in assumptions. Pure assertions in \(\Gamma\) have type \(\text{Prop}\) in Coq and correspond to Coq propositional assumptions.

Two different kinds of judgments correspond to two different kinds of conclusions. The Coq proposition “derives \(P\) \(Q\)” or in notation “\(P \vdash Q\)”, represents a separation logic entailment.²

The Hoare triple \(\{P\} c \{Q, [\vec{R}]\}\) is also defined as a Coq proposition: “\(\text{semax} \ \Delta \ P \ c \ (Q, [\vec{R}])\)”. In the real Coq formalization, \(\Delta\) records not only function specifications but also variable types of the C program \(c\).

²Verifiable C treats \(P \vdash Q\) as a primitive judgment and treats \(\vdash P\) as an abbreviation of \(\top \vdash P\). This is logically equivalent with the language this thesis uses: taking \(\vdash P\) as primitive and defining \(P \vdash Q\) by \(\vdash P \rightarrow Q\).
The following examples demonstrate this correspondence between judgments and Coq proof goals.

Judgment:

\[ p; q; v; \quad p \neq q; \quad p \mapsto v \land q \mapsto v \vdash \bot \]

Coq Proof Goal:

\[
\begin{align*}
& p: \text{val} \\
& q: \text{val} \\
& v: \text{val} \\
& H: p \neq q \\
& \hline
& p \mapsto v \land q \mapsto v \vdash \bot
\end{align*}
\]

Judgment:

\[ x; \cdots; \vdash \{ [x] = x \land \text{emp} \} \quad y = x \{ [y] = x \land \text{emp}, [\perp] \} \]

Coq Proof Goal:

\[
\begin{align*}
& x: \text{val} \\
& \Delta := \ldots \text{ (* Definition of the C-language context *)} \\
& \hline
& \text{semax} \Delta \\
& \quad ([x] = x \land \text{emp}) \ (y = x) \ ([y] = x \land \text{emp}, [\perp])
\end{align*}
\]
2.6 Proof rules

The proof rules for entailments can be mainly classified into the following categories: two special proof rules for assumptions (ASSU and WEAKEN) and proof rules for propositional connectives, for quantifiers, for separation logic, for domain-specific theories and for C-language-specific expression evaluation.

2.6.1 Generic proof rules for entailments

Figure 2.1 and figure 2.2 list the generic proof rules. Here, an assertion is called nonspatial if every predicate in it is nonspatial.

It is known that these proof rules for propositional connectives form intuitionistic propositional logic [46]. Based on that, adding the rules for quantifiers forms intuitionistic first order logic (or intuitionistic higher order, if the quantifiers can quantify over functions and predicates). In this thesis, I will call intuitionistic propositional (first order, higher order) logic plus the rules for separating connectives (excluding *-\&-COMM) minimum propositional (first order, higher order) separation logic.
Proof Rules for Assumptions:

**ASSU** If $P_{\text{Pure}} \in \Gamma$, then $\Sigma; \Gamma \vdash P_{\text{Pure}}$

**WEAKEN** If $\Gamma_1 \subseteq \Gamma_2$ and $\Sigma; \Gamma_1 \vdash P$, then $\Sigma; \Gamma_2 \vdash P$

Proof Rules for Propositional Connectives:

**MP:** If $\Sigma; \Gamma \vdash P$ and $\Sigma; \Gamma \vdash P \to Q$ then $\Sigma; \Gamma \vdash Q$

$\to_1$: $\Sigma; \Gamma \vdash P \to (Q \to P)$

$\to_2$: $\Sigma; \Gamma \vdash (P \to Q \to R) \to (P \to Q) \to (P \to R)$

$\land$-**INTRO:** $\Sigma; \Gamma \vdash P \to Q \to P \land Q$

$\land$-**ELIM:** $\Sigma; \Gamma \vdash P \land Q \to P$, $\Sigma; \Gamma \vdash P \land Q \to Q$

**$\lor$-ELIM:** $\Sigma; \Gamma \vdash (P \to R) \to (Q \to R) \to P \lor Q \to R$

$\lor$-**INTRO:** $\Sigma; \Gamma \vdash P \to P \lor Q$, $\Sigma; \Gamma \vdash Q \to P \lor Q$

$\bot$-**INTRO:** $\Sigma; \Gamma \vdash \bot \to P$

Proof Rules for Quantifiers:

$\forall$-**INTRO:** If $\Sigma; x; \Gamma; P \vdash Q$ and $x$ does not freely occur in $\Gamma$ or $P$ then $\Sigma; \Gamma; P \vdash \forall x. Q$

$\forall$-**ELIM:** $\Sigma; \Gamma; \forall x. P \vdash P[E/x]$

$\exists$-**ELIM:** If $\Sigma; x; \Gamma; P \vdash Q$ and $x$ does not freely occur in $\Gamma$ or $Q$ then $\Sigma; \Gamma; \exists x. P \vdash Q$

$\exists$-**INTRO:** $\Sigma; \Gamma; P[E/x] \vdash \exists x. P$

Figure 2.1: Generic Proof Rules for Entailments (Part 1)
Proof Rules for Propositional Separation Logic:

\(-\text{comm}\):
\[ \Sigma; \Gamma; P \ast Q \vdash Q \ast P \]

\(-\text{assoc}\):
\[ \Sigma; \Gamma; P \ast (Q \ast R) \nvdash (P \ast Q) \ast R \]

\(-\text{emp}\):
\[ \Sigma; \Gamma; P \ast \text{emp} \nvdash P \]

\text{ADJOINT}1:
If \( \Sigma; \Gamma; P \vdash Q \rightarrow R \) then \( \Sigma; \Gamma; P \ast Q \vdash R \)

\text{ADJOINT}2:
If \( \Sigma; \Gamma; P \ast Q \vdash R \) then \( \Sigma; \Gamma; P \vdash Q \rightarrow R \)

\(-\land-\text{comm}\):
If \( P \) is nonspatial
then \( \Sigma; \Gamma; P \land (Q \ast R) \nvdash (P \land Q) \ast P \)

Figure 2.2: Generic Proof Rules for Entailments (Part 2)

From this set of proof rules, we can derive all intuitionistic tautologies. And we can also derive many useful proof rules for separation logic:

\text{PURE-ELIM}:
If \( \Sigma; \Gamma; P^{\text{Pure}}; P \vdash Q \), then \( \Sigma; \Gamma; P^{\text{Pure}} \land P \vdash Q \)

If \( \Sigma; \Gamma; P_{1} \vdash Q_{1} \) and

\text{SEPCON-MONO}:
\[ \Sigma; \Gamma; P_{2} \vdash Q_{2} \]
then \( \Sigma; \Gamma; P_{1} \ast P_{2} \vdash Q_{1} \ast Q_{2} \)

If \( \Sigma; \Gamma; Q_{1} \vdash P_{1} \) and

\text{WAND-MONO}:
\[ \Sigma; \Gamma; P_{2} \vdash Q_{2} \]
then \( \Sigma; \Gamma; P_{1} \ast P_{2} \vdash Q_{1} \ast Q_{2} \)

\text{WAND-MP}:
\[ \Sigma; \Gamma; P \ast (P \ast Q) \vdash Q \]

Figure 2.3: Derived Rules for Separation Logic Entailments
2.6.2 Domain-specific proof rules for entailments

Different domain-specific theories have their own proof rules. I cannot enumerate all of them here. Figure 2.4 shows some examples of domain-specific proof rules and C expression evaluation rules, for example, the theory of arithmetic, the theory of equality, the theory of lists (including structural induction over lists) etc.

Proof Rules for Nonspatial Predicates:

\[
\Sigma; \Gamma \vdash x < x + 1
\]

\[
\Sigma; \Gamma; E_1 = E_2 \vdash P[E_1/x] \leftrightarrow P[E_2/x]
\]

\[
\Sigma; \Gamma \vdash (\lambda x. E_1)(E_2) = E_1[E_2/x]
\]

If \( \Sigma; \Gamma; P \vdash Q[\text{nil}/x] \) and

\[
\Sigma; \Gamma; P \vdash \forall a, l. Q[l/x] \rightarrow Q[\text{cons}(a, l)/x]
\]

then \( \Sigma; \Gamma; P \vdash \forall l. Q[l/x] \)

Proof Rules for Spatial Predicates:

\[
\Sigma; \Gamma; p \mapsto x \ast p \mapsto y \vdash \bot
\]

Proof Rules for C Expression Evaluation:

If \( x \) is a nonaddressable variable of type \texttt{unsigned int},

then \( \Sigma; \Gamma \vdash [x + 1] = ([x] + 1) \mod (\text{max unsigned} + 1) \)

If \( x \) is a nonaddressable variable of type \texttt{signed int},

then \( \Sigma; \Gamma \vdash [x] + 1 \leq \text{max unsigned} \vdash [x + 1] = [x] + 1 \)

Figure 2.4: Examples of domain-specific or language-specific proof rules
Linked lists are the most well-known examples in separation logic. I list them separately here: Figure 2.5 demonstrates proof rules for two related predicates—linked lists (listrep(p, l) means that l is stored as a linked list with head pointer p) and linked list segments (lseg_rec(p, q, l) means that l is stored as a linked list segment with head pointer p and tail pointer q).

$$
\Sigma; \Gamma; \text{listrep}(p, \text{nil}) \vdash p = \text{null} \land \text{emp}
$$

$$
\Sigma; \Gamma; \text{listrep}(p, \text{cons}(a, l)) \vdash \exists q. p \mapsto a \ast p + 4 \mapsto q \ast \text{listrep}(q, l)
$$

(assuming 4 bytes per word)

$$
\Sigma; \Gamma; \text{lseg_rec}(p, q, \text{nil}) \vdash p = q \land \text{emp}
$$

$$
\Sigma; \Gamma; \text{lseg_rec}(p, q, \text{cons}(a, l)) \vdash
\exists p'. p \mapsto a \ast p + 4 \mapsto p' \ast \text{lseg_rec}(p', q, l)
$$

$$
\Sigma; \Gamma; \text{lseg_rec}(p, \text{null}, l) \vdash \text{listrep}(p, l)
$$

$$
\Sigma; \Gamma; \text{lseg_rec}(p, q, l_1 \cdot l_2) \vdash \exists p'. \text{lseg_rec}(p, p', l_1) \ast \text{lseg_rec}(p', q, l_2)
$$

Figure 2.5: Proof rules for linked lists
2.6.3 Proof rules for Hoare triples

Besides those proof rules for entailments, Figure 2.6 shows proof rules for generic separation Hoare logic. Figure 2.7 shows some proof rule examples for C-specific separation Hoare logic.

**HOARE-SEQ:**

If $\Sigma; \Gamma; \Delta \vdash \{ P \} c_1 \{ Q, \vec{s} \}$ and

$$\Sigma; \Gamma; \Delta \vdash \{ Q \} c_2 \{ R, \vec{s} \}$$

then $\Sigma; \Gamma; \Delta \vdash \{ P \} c_1; c_2 \{ R, \vec{s} \}$.

**HOARE-CON:**

If $\Sigma; \Gamma; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \}$,

$$\Sigma; \Gamma; P' \vdash P, \Sigma; \Gamma; Q \vdash Q'$$ and for any $i$, $\Sigma; \Gamma; R_i \vdash R'_i$

then $\Sigma; \Gamma; \Delta \vdash \{ P' \} c \{ Q', [\vec{R'}] \}$.

**HOARE-EXISTS:**

If $\Sigma; x; \Gamma \vdash \{ P \} c \{ Q, [\vec{R}] \}$ and $x$ does not freely occur in $\Gamma$, $Q$ or $\vec{R}$

then $\Sigma; \Gamma; \Delta \vdash \{ \exists x. P \} c \{ Q, [\vec{R}] \}$.

**HOARE-PURE:**

If $\Sigma; \Gamma; P^{\text{Pure}}; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \}$

then $\Sigma; \Gamma; \Delta \vdash \{ P^{\text{Pure}} \land P \} c \{ Q, [\vec{R}] \}$.

**FRAME:**

If $\Sigma; \Gamma; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \}$ and any nonaddressable variable which is modified in $c$ does not freely occur in $F$

then $\Sigma; \Gamma; \Delta \vdash \{ P \ast F \} c \{ Q \ast F, [\vec{R} \ast F] \}$.

---

The restriction on nonaddressable program variables’ occurrence in rule FRAME looks very C-language specific. But actually, in most separation Hoare logics, there are local variables whose values are not part of separating resources. They are similar to nonaddressable variables of C in this thesis.
If \( x \) is a nonaddressable variable, \( x \) and \( e \) have the same C scalar type and

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \ x = e \left\{ [x] = a \land \exists x'. P[x'/x], [\top] \right\}.
\]

If \( x \) is a nonaddressable variable, \( x \) and \( e \) have the same C scalar type and

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \ x = e \left\{ [x] = a \land \exists x'. P[x'/x], [\top] \right\}.
\]

If \( e_1 \) and \( e_2 \) has the same C scalar type and

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \ x = e_1 \left\{ P \land P \implies a \implies b, [\top] \right\}.
\]

If \( \Sigma; \Gamma; \Delta \vdash b \downarrow \), \( \Sigma; \Gamma; \Delta \vdash \{ [b] = true \land P \} \ c_1 \left\{ Q, [R] \right\} \) and

\[
\Sigma; \Gamma; \Delta \vdash \{ [b] = false \land P \} \ c_2 \left\{ Q, [R] \right\}.
\]

If \( \Sigma; \Gamma; \Delta \vdash \{ I \} \ c_1 \left\{ I_{con}, [Q, I_{con}, Q_{ret}] \right\} \) and

\[
\Sigma; \Gamma; \Delta \vdash \{ I \} \ c \left\{ I_{brk}, [Q_{brk}, Q_{con}, Q_{ret}] \right\}.
\]

**SEMAX-LOAD:**

\[
\Sigma; \Gamma; P \vdash [\&e] = p \land (p \mapsto a \ast \top) \land \& e \downarrow
\]

then \( \Sigma; \Gamma; \Delta \vdash \{ P \} \ x = e \left\{ [x] = a \land \exists x'. P[x'/x], [\top] \right\}.
\]

**SEMAX-STORE:**

\[
\Sigma; \Gamma; P \ast p \mapsto a \vdash [\&e] = p \land [e_1] = b \land \& e_1 \downarrow \land e_2 \downarrow
\]

then \( \Sigma; \Gamma; \Delta \vdash \{ P \ast p \mapsto a \} \ e_1 = e_2 \left\{ P \ast p \mapsto b, [\top] \right\}.
\]

If \( \Sigma; \Gamma; \Delta \vdash \{ R \ast P(a)[\bar{v}/\overline{\bar{v}}] \} \ x = f(\bar{v}) \left\{ R \ast Q(a)[[x]/ret_val], [\top] \right\}.
\]

**SEMAX-CALL:**

\[
\Sigma; \Gamma; \Delta \vdash \{ R \ast P(a)[\bar{v}/\overline{\bar{v}}] \} \ x = f(\bar{v}) \left\{ R \ast Q(a)[[x]/ret_val], [\top] \right\}.
\]

**SEMAX-IF:**

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \ x = e_1 \left\{ Q, [R] \right\} \) and
\]

**SEMAX-LOOP:**

\[
\Sigma; \Gamma; \Delta \vdash \{ I \} \ c \left\{ I_{con}, [Q, I_{con}, Q_{ret}] \right\} \) and
\]

**Figure 2.7:** Proof rules for C-specific separation Hoare logic
2.7 Coq formalization of proof rules

Generally speaking, the proof rules shown here (for entailments and for Hoare logic) are slightly different from their statements in Coq. But they correspond precisely to applying such Coq theorems instead. For example, the following shows the sequence rule (presented in the previous section) and its Coq formalization.

\[
\text{If } \Sigma; \Gamma; \Delta \vdash \{P\} \ c_1 \ \{Q, [\vec{S}]\} \text{ and } \Sigma; \Gamma; \Delta \vdash \{Q\} \ c_2 \ \{R, [\vec{S}]\} \text{ then } \Sigma; \Gamma; \Delta \vdash \{P\} \ c_1; c_2 \ \{R, [\vec{S}]\}. 
\]

**Theorem** semax-seq: \(\forall \Delta \ P \ Q \ R \ \vec{S} \ c_1 \ c_2,\)

\[
\text{semax } \Delta \ P \ c_1 \ (Q, [\vec{S}]) \rightarrow \\
\text{semax } \Delta \ Q \ c_2 \ (R, [\vec{S}]) \rightarrow \\
\text{semax } \Delta \ P \ (c_1; c_2) \ (R, [\vec{S}]).
\]

Here, the logical context \(\Sigma\) and assumptions \(\Gamma\) do not appear in the Coq theorem. But applying this Coq theorem (i.e. using the Coq tactic: `apply (semax_seq P Q R)` will reduce the proof goal in Figure 2.8 to the proof goals in Figures 2.9 and 2.10—they correspond to the conclusion and the assumptions of the math statement of HOARE-SEQ.

\[
\begin{array}{lll}
\Sigma & \Gamma & \Sigma \\
\hline
\Delta & \text{semax } \Delta \ P \ (c_1; c_2) \ (R, [\vec{S}]) & \Sigma \\
\hline
\end{array}
\]

Figure 2.8: Original goal

\[
\begin{array}{lll}
\Sigma & \Gamma & \Sigma \\
\hline
\Delta & \text{semax } \Delta \ P \ c_1 \ (Q, [\vec{S}]) & \Sigma \\
\hline
\end{array}
\]

Figure 2.9: Reduced goal

\[
\begin{array}{lll}
\Sigma & \Gamma & \Sigma \\
\hline
\Delta & \text{semax } \Delta \ Q \ c_2 \ (R, [\vec{S}]) & \Sigma \\
\hline
\end{array}
\]

Figure 2.10: Reduced goal
2.8 Semantics and soundness

Verifiable C defines an operational semantics of C, which is a proved-equivalent variant of CompCert Clight [42, 43]. All proof rules mentioned above (except domain-specific ones) are proved sound with respect to this operational semantics, a step-indexed C memory model and a step-indexed interpretation of Hoare triple validity by Appel et al. [5].

This thesis focuses on using these proved sound rules in verification. Thus, I choose to omit Verifiable C’s semantics and soundness proof here since they are less relevant and not practical to present in full here.
2.9 Hoare logic proofs vs. decorated programs

In this thesis, I will mostly use decorated programs to represent C program correctness proofs. But actually, it is Hoare logic that backs up these proofs. For example: Figure 2.11 is a decorated program which proves:

\[
\begin{align*}
\text{a; b; } \vdash & \{ [x] = a \wedge [y] = b \} \\
& z = x; x = y; y = z; \\
& \{ [x] = b \wedge [y] = a \}.
\end{align*}
\]

The formal Hoare logic proof behind it is a series of application of \textsc{semax-set}, \textsc{hoare-seq}, \textsc{hoare-con} and first order logic, see Figure 2.12.

Using assertions as comments in programs to illustrate program correctness is a well-known technique and has already been widely used in software development. In comparison, decorated C programs are easier to understand. Writing the corresponding Hoare logic proofs is tedious. Therefore I will use decorated C programs to illustrate proofs. And in Part II of this thesis, one task of VST-Floyd is exactly to let users complete Hoare logic proofs in Coq as if they are writing a decorated program from the top down.

\begin{verbatim}
1 \{ [x] = a \wedge [y] = b \}
2 z = x;
3 \{ [z] = a \wedge \exists z. [x] = a \wedge [y] = b \}
4 \{ [z] = a \wedge [x] = a \wedge [y] = b \}
5 x = y;
6 \{ [x] = b \wedge \exists x. [z] = a \wedge x = a \wedge [y] = b \}
7 \{ [x] = b \wedge [z] = a \wedge [y] = b \}
8 y = z;
9 \{ [y] = a \wedge \exists y. [x] = b \wedge [z] = a \wedge y = b \}
10 \{ [x] = b \wedge [y] = a \}
\end{verbatim}

Figure 2.11: Example: Decorated C Program
Figure 2.12: Example: Hoare Logic Proof

30
Part I

Magic wand as frame
In this part, I demonstrate magic-wand-as-frame, a disciplined way of using magic wand in separation logic based program verification. I begin with an overview of other authors’ previous work using magic wand (chapter 3). In general, using magic wand (in an undisciplined way) in program verification is complicated—even determining the soundness a magic-wand-involved entailment can be tricky (chapter 4). But verification by magic-wand-as-frame does not have this problem.

I introduce the proof rules of magic-wand-as-frame in chapter 5 and illustrate the disciplines by two program verification examples afterwards (chapter 6 and 7). I also use magic-wand-as-frame to improve “ramification”, a closely related research result about separation logic (chapter 8).
Chapter 3

Previous work using magic wand

Magic wand, the separating implication, is not widely used previously in program verification.

In the early days of separation logic, magic wand was used to generate weakest preconditions and verification conditions for automated program verification. However, those verification conditions are not human readable or understandable, and decision procedures for entailment checking with magic wand are quite complex.

Moreover, Brotherston [16] proved that the provability of classical propositional separation is undecidable. In other words, any decision procedure for testing the provability of magic-wand-involved expressions must be domain specific.

Magic wand is also unwelcome in interactive program verification or pencil-and-paper proofs. Authors tend to use forward verification instead of backward verification since it is easier to understand a program correctness proof that goes in the same direction as program execution. “Forward” Hoare logic rules do not generate magic wand expressions; therefore, most authors find that the expressive power of star is already strong enough. For example, we need to define separation logic predicates for different data structures (like records, arrays, linked list and binary trees) in order to verify related programs. Berdine et al. [10] and Charguéraud [18] show that these
predicates can be defined with separating conjunction only. In comparison, authors like Tuerk [51] believe that using magic wand is very complicated in verifying binary search tree operations.

The first known practical application of magic wand in program verification is Yang’s correctness proof [53] of the Schorr-Waite graph marking algorithm. The assertions used in his proof deeply mix conjunctions and separating conjunctions together. Specifically, his assertions (especially the loop invariant) have form

\[(P_0,0 \ast P_0,1 \ast \ldots) \land (P_1,0 \ast P_1,1 \ast \ldots) \land \ldots \land (P_n,0 \ast P_n,1 \ast \ldots)\]

in which \(P_{i,j}\) are all spatial. This mixed usage of separating conjunction and normal conjunction was later abandoned by the verification community since those proofs (e.g. Yang’s proof) are too complicated. Authors now prefer to use assertions with the following form:

\[(P_0 \land P_1 \land \ldots \land P_n) \land (Q_0 \ast Q_1 \ast \ldots \ast Q_m)\]

Here, \(P_i\) are all nonspatial and \(Q_i\) are all spatial.

Some previous authors found separating implication is useful for tracking a local piece of memory while still remembering what the global ones satisfies, as long as the local part is unchanged. Charguéraud [18] mentioned that binary-search-tree look-up can be verified functionally correct using magic wand. In such look-up operations, the binary search tree is totally unmodified. Maeda et al. [44] showed application of magic wand in verifying memory safety of linked-list traverse, linked-list append, FIFO queue enqueue and binary search tree delete. Although the sub-linked-lists or subtrees are actually modified, the local portion before and after the modification will satisfy the same separation logic assertion in shape analysis. Their examples either had no modification operation or could only support shape analysis.
Also, previous authors noticed that magic wand can be used to describe the global effect of a local modification. The backward store rule is such an example:

**STORE-BACKWARD:**

If \( e_1 \) and \( e_2 \) have the same C scalar type then

\[
\Sigma; \Gamma; \Delta \vdash \begin{cases}
[&e_1] = p \land [e_2] = b \land &e_1 \Downarrow \land e_2 \Downarrow \land \\
p \mapsto a \ast (p \mapsto b \rightharpoonup P)
\end{cases}
\]

\( e_1 = e_2 \left\{ P, [\overline{1}] \right\} \)

This rule showed up in one of the earliest papers of separation logic \[48\] (I put the C version here, not the original version).

Jensen, Birkedal and Sestoft \[34\] use magic wand to relate a local update to a global update in a linked-list-with-views example. They use quantifiers to characterize arbitrary local updates in the future. Hobor and Villard \[30\] state a general proof rule \textsc{ramif} which connects local and global portions:

**RAMIF:**

If \( \Sigma; \Gamma; \Delta \vdash \{ P_L \} \left\{ Q_L, [\overline{1}] \right\} \), \( \Sigma; \Gamma; P_G \vdash P_L \ast (Q_L \rightharpoonup Q_G) \) and any nonaddressable variable which is modified in \( c \) does not freely occur in \( (Q_L \rightharpoonup Q_G) \)

then \( \Sigma; \Gamma; \Delta \vdash \{ P_G \} \left\{ Q_G, [\overline{1}] \right\} \)

Iris \[37\] uses magic wand and modalities to formalize Hoare triples. But that is more about formalizing program logic rather than using program logics in program verification. I will discuss this related work in Part II.

Besides separating conjunction and separating implication, some separation logic articles also use another connective: septraction. It is also called existential magic wand and is written as “\( \lnot \rightarrow \)" or “\( \lnot \bullet \)". In short, a program state \( s \) satisfies \( P \rightarrow Q \) if
and only if there exists $s_1$ and $s_2$ such that $s \oplus s_1 = s_2$, $s_1$ satisfies $P$ and $s_2$ satisfies $Q$. For classical separation logics, $\neg \circ$ can be treated as the dual of $\neg \ast$. In other words, $P \neg \circ Q$ is logically equivalent with $\neg(P \neg \ast \neg Q)$. For nonclassical logics, there is no conclusive research result regarding septraction’s proof theory.

Additionally, many separation logic entailments about “$\neg \circ$” can be rewritten using “$\neg \ast$”. For example, $(L \neg \circ G) \ast L' \vdash G$ is used in RGSep [52] to express: $G$ is stable under a local action from $L$ to $L'$. But it is known that $(L \neg \circ G) \ast L' \vdash G'$ is almost the same as $G \vdash L \ast (L' \neg \ast G')$. Specifically, when $L$ is precise, $G \vdash L \ast (L' \neg \ast G')$ implies $(L \neg \circ G) \ast L' \vdash G'$. When $G \vdash L \ast \top$, $(L \neg \circ G) \ast L' \vdash G'$ implies $G \vdash L \ast (L' \neg \ast G')$. In real verifications, these two side conditions “$L$ is precise” and “$G \vdash L \ast \top$” are usually true. In this thesis, I will only focus on the application of separating implication, “$\neg \ast$”. 

36
Chapter 4

Discussion: the limited proof power

In separation logic, separating conjunction is like addition—commutative and associative—but magic wand is not totally like subtraction. For example, \( x \) and \((x + y) - y\) are equal in arithmetic, but the analogous wand-using separation logic formulae, \( P \) and \( Q \rightarrow^* (P \ast Q) \), are not equivalent in logic:

\[
\Sigma; \Gamma; P \vdash Q \rightarrow^* (P \ast Q)
\]

\[
\Sigma; \Gamma; Q \rightarrow^* (P \ast Q) \not\vdash P
\]

Thus, we cannot treat \( Q \rightarrow^* (P \ast Q) \) as describing the remainder left after subtracting a piece of memory that satisfies \( Q \) from a larger piece of memory that satisfies \( P \ast Q \).

Generally speaking, determining the soundness of a magic-wand-involved proof rule is very tricky. This chapter discusses the most typical ones. To show a rule sound, I will derive it from the basic rules in Figure 2.1 and 2.2. To show a rule unsound, I will pick out an instance of it and demonstrate that it is invalid on some model.
4.1 A simple semantics

I will use the following semantics to demonstrate unsoundness.

Given a mapping $J$ from logical variables to their values ($\mathbb{N}$), a heap $H$ which is a partial function from addresses ($\mathbb{N}$) to numbers ($\mathbb{N}$) and a separation logic assertion $P$, I define the satisfaction relation $J, H \models P$ as follows.

$J, H \models E_1 = E_2$ if and only if $\text{eval}^J(E_1) = \text{eval}^J(E_2)$

$J, H \models \text{emp}$ if and only if $\text{dom}(H) = \{\}$

$J, H \models E_1 \mapsto E_2$ if and only if $\text{dom}(H) = \{\text{eval}^J(E_1)\}$ and $H(\text{eval}^J(E_1)) = \text{eval}^J(E_2)$

$J, H \models P \land Q$ if and only if $J, H \models P$ and $J, H \models Q$

$J, H \models P \lor Q$ if and only if there exists $H_1$ and $H_2$ such that

$$\oplus(H_1, H_2, H), J, H_1 \models P \text{ and } J, H_2 \models Q$$

$J, H \models P \rightarrow Q$ if and only if for any $H_1$ and $H_2$

if $\oplus(H_1, H, H_2)$ and $J, H_1 \models P$, then $J, H_2 \models Q$

$J, H \models \exists x : \tau. P$ if and only if there exists $a$ of type $\tau$, such that $J[a/x], H \models P$

$J, H \models \text{listrep}(E_1, E_2)$ if and only if

$J, H \models E_1 = \text{null} \land E_2 = \text{nil} \land \text{emp}$ or

$J, H \models \exists x, y, z. E_2 = \text{cons}(x, z) \land E_1 \mapsto x \land E_1 + 4 \mapsto y \land \text{listrep}(y, z)$

$J, H \models \text{lseg} \text{rec}(E_1, E_2, E_3)$ if and only if

$J, H \models E_1 = E_2 \land E_3 = \text{nil} \land \text{emp}$ or

$J, H \models \exists x, y, z. E_3 = \text{cons}(x, z) \land E_1 \mapsto x \land E_1 + 4 \mapsto y \land \text{lseg} \text{rec}(y, E_2, z)$
Here, \( \text{eval}^J(E) \) represents the evaluation of expression \( E \) given variables \( J \), \( \text{dom}(H) \) is the domain of heap \( H \) and \( \oplus(H_1, H_2, H_3) \) means that \( H_3 \) is the disjoint union of \( H_1 \) and \( H_2 \). Also, later in this chapter, I will use \([p_1 \mapsto v_1; p_2 \mapsto v_2; \ldots; p_n \mapsto v_n]\) to represent a heap with address \( p_1, p_2, \ldots, p_n \) and have values \( v_1, v_2, \ldots, v_n \) stored in these addresses respectively. I use the same notation “\( \mapsto \)” in heap-representation and in the assertion language. But usually, it is unambiguous from the context.

For the sake of space, I only list the semantics of a subset of connectives and predicates. I will only use these connectives and predicates in the unsoundness proofs in this chapter.

### 4.2 Distinguish sound rules from unsound rules

**Theorem 1.** \( \Sigma; \Gamma; Q \rightsquigarrow (P \ast Q) \vdash P \) is unsound.

*Proof.* First of all, there are no heaps \( m \) and \( n \) such that:

\[
\oplus([0x10 \mapsto 0], m, n) \quad \text{and} \quad m \models 0x10 \mapsto 0.
\]

Thus, \([0x10 \mapsto 0] \models 0x10 \mapsto 0 \not\rightarrow (0x10 \mapsto 0 \ast 0x14 \mapsto 0)\).

However, \([0x10 \mapsto 0] \not\models 0x14 \mapsto 0\).

So, \( 
\vdots; 0x10 \mapsto 0 \not\rightarrow (0x10 \mapsto 0 \ast 0x14 \mapsto 0) \not\models 0x14 \mapsto 0. \)

Thus, \( \Sigma; \Gamma; Q \rightsquigarrow (P \ast Q) \vdash P \) is unsound. \( \square \)

**Theorem 2.** \( \Sigma; \Gamma; 0x10 \mapsto 0 \not\rightarrow (0x14 \mapsto 0 \ast P) \vdash 0x14 \mapsto 0 \ast \top \) is unsound.

*Proof.* Similarly, \([0x10 \mapsto 0] \models 0x10 \mapsto 0 \not\rightarrow (0x10 \mapsto 0 \ast 0x14 \mapsto 0)\). But \([0x10 \mapsto 0] \not\models 0x14 \mapsto 0 \ast \top\). \( \square \)

These two theorems show that almost no internal structure of subheaps can be revealed by a single magic wand expression. The only exception is about emp.
Theorem 3. \( \Sigma; \Gamma; \text{emp} \not\rightarrow P \vdash P \) is sound.

Proof. By \(*\)-emp, \( \Sigma; \Gamma; \text{emp} \not\rightarrow P \vdash \text{emp} *(\text{emp} \not\rightarrow P) \).

By wand-MP, \( \Sigma; \Gamma; \text{emp} *(\text{emp} \not\rightarrow P) \vdash P \).

Thus \( \Sigma; \Gamma; \text{emp} \not\rightarrow P \vdash P \).

This proof shows that useful information can be derived if feeding a magic-wand expressions with a separating conjunct identical with its antecedent. The following proof rule is another instance.

Theorem 4. \( \Sigma; \Gamma; P(E) \not\rightarrow \forall x . (P(x) \not\rightarrow Q) \vdash Q \) is sound.

Proof. By \( \forall\)-elim, \( \Sigma; \Gamma; \forall x . (P(x) \not\rightarrow Q) \vdash P(E) \not\rightarrow Q \).

By sepcon-mono, \( \Sigma; \Gamma; P(E) \not\rightarrow \forall x . (P(x) \not\rightarrow Q) \vdash P(E) \not\rightarrow (P(E) \not\rightarrow Q) \).

By wand-MP, \( \Sigma; \Gamma; P(E) \not\rightarrow (P(E) \not\rightarrow Q) \vdash Q \).

Thus, \( \Sigma; \Gamma; P(E) \not\rightarrow \forall x . (P(x) \not\rightarrow Q) \vdash Q \).

The feeding separating conjunct can even be different from the antecedent.

Theorem 5. \( \Sigma; \Gamma; 0x10 \mapsto 0 \not\rightarrow (0x10 \mapsto 1 \not\rightarrow 0x10 \mapsto 1 \not\rightarrow Q) \vdash 0x10 \mapsto 0 \not\rightarrow Q \) is valid with respect to the semantics defined in section 4.1.

Proof. For any subheap \( n \), if \( n \models 0x10 \mapsto 0 \not\rightarrow (0x10 \mapsto 1 \not\rightarrow 0x10 \mapsto 1 \not\rightarrow Q) \), then there exists \( m \) such that

\[
\bigoplus ([0x10 \mapsto 0], m, n) \tag{4.1}
\]

\( m \models 0x10 \mapsto 1 \not\rightarrow 0x10 \mapsto 1 \not\rightarrow Q. \tag{4.2}
\]

From (4.1), there must exist \( n' \) such that \( \bigoplus ([0x10 \mapsto 1], m, n') \).

But from (4.2), for any \( r \), if \( \bigoplus ([0x10 \mapsto 1], m, r) \), then \( r \models 0x10 \mapsto 1 \not\rightarrow Q \).

\[^1\text{This rule could be more formally stated as follows: if } x \text{ does not freely occur in } Q, \text{ then } \Sigma; \Gamma; P[E/x] \not\rightarrow \forall x . (P \not\rightarrow Q) \vdash Q. \]
So, actually, \( n' \models 0x10 \mapsto 1 \ast Q \). Thus, \( m \models Q \).

Together with (4.1), we get \( n \models 0x10 \mapsto 0 \ast Q \).

We have seen that deriving facts from magic wand expressions can be hard; but actually, establishing magic wand can also be very tricky. Yang uses the following rule in his correctness proof \([53]\) of Schorr-Waite graph marking algorithm.

**Theorem 6.** \( \Sigma; \Gamma; P \land (0x10 \mapsto 0 \ast \top) \models 0x10 \mapsto 0 \ast (0x10 \mapsto 0 \ast P) \) is valid with respect to the semantics defined in section 4.1.

**Proof.** For any subheap \( n \), if \( n \models P \land (0x10 \mapsto 0 \ast \top) \), then there exists \( m \) such that

\[
\oplus ([0x10 \mapsto 0], m, n) \tag{4.3}
\]

\[
n \models P. \tag{4.4}
\]

By (4.3), the only result of extending \( m \) with \([0x10 \mapsto 0]\) is \( n \).

Together (4.4), \( m \models 0x10 \mapsto 0 \ast P \).

Thus, \( n \models 0x10 \mapsto 0 \ast (0x10 \mapsto 0 \ast P) \). \( \square \)

Krishnaswami \([39]\) verifies data structures (like linked list, binary search tree and other collectors) with two or more iterators. He uses the following proof rule in his verification. However, it is unsound.

**Theorem 7.** Let \( \text{iter}(p, i, l) \) be the abbreviation of

\[
\exists q, l_1, l_2. l = l_1 \cdot l_2 \land \text{lseg.rec}(p, q, l_1) \ast \text{listrep}(q, l_2) \ast i \mapsto q.
\]

Then \( \Sigma; \Gamma; \text{iter}(p, i, l) \models \text{listrep}(p, l) \ast (\text{listrep}(p, l) \ast \text{iter}(p, i, l)) \) is unsound.

This rule is seemingly correct. In order to prove it, a natural approach is to extract the existentially quantified \( l_1 \) and \( l_2 \) on the left side and instantiate the corresponding existential on the right side. In other words, it suffices to prove the following
entailment sound.

\[ \Sigma; \Gamma; \exists q. \text{lseg}_\text{rec}(p,q,l_1) \ast \text{listrep}(q,l_2) \ast i \mapsto q \vdash \text{listrep}(p,l_1 \cdot l_2) \ast (\text{listrep}(p,l_1 \cdot l_2) \ast \exists q. \text{lseg}_\text{rec}(p,q,l_1) \ast \text{listrep}(q,l_2) \ast i \mapsto q) \]  

(4.5)

Since we know \( \Sigma; \Gamma; \text{listrep}(p,l_1 \cdot l_2) \not\models \exists q. \text{lseg}_\text{rec}(p,q,l_1) \ast \text{listrep}(q,l_2) \), the entailment (4.5) is like a variant of \( \Sigma; \Gamma; Q \models P \Rightarrow P \ast Q \), which is obviously sound by \text{wand-adjoint}. However, the additional existential quantifiers in (4.5) are actually problematic.

**Proof.** Let \( m, m', n, n' \) be heaps:

\[
\begin{align*}
m &= [0x10 \mapsto 1; 0x14 \mapsto 0x20; 0x20 \mapsto 2; 0x24 \mapsto \text{null}; 0x80 \mapsto 0x20] \\
n &= [0x10 \mapsto 1; 0x14 \mapsto 0x20; 0x20 \mapsto 2; 0x24 \mapsto \text{null}] \\
m' &= [0x10 \mapsto 1; 0x14 \mapsto 0x30; 0x30 \mapsto 2; 0x34 \mapsto \text{null}; 0x80 \mapsto 0x20] \\
n' &= [0x10 \mapsto 1; 0x14 \mapsto 0x30; 0x30 \mapsto 2; 0x34 \mapsto \text{null}]
\end{align*}
\]

First, \( m' \not\models \text{iter}(0x10, 0x80, [1;2]) \).

Second, \( n' \models \text{listrep}(0x10, [1;2]) \).

So, \( [0x80 \mapsto 0x20] \not\models \text{listrep}(0x10, [1;2]) \Rightarrow \text{iter}(0x10, 0x80, [1;2]) \) since \( \oplus(n', [0x80 \mapsto 0x20], m') \).

Third, \( n \) is the only subheap in \( m \) such that \( n \models \text{listrep}(0x10, [1;2]) \).

Also, \( \oplus(n, [0x80 \mapsto 0x20], m) \).

Thus, \( m \not\models \text{listrep}(0x10, [1;2]) \ast (\text{listrep}(0x10, [1;2]) \Rightarrow \text{iter}(0x10, 0x80, [1;2])) \).

But obviously, \( m \models \text{iter}(0x10, 0x80, [1;2]) \).

That shows the unsoundness of

\[ \Sigma; \Gamma; \text{iter}(p,i,l) \vdash \text{listrep}(p,l) \ast (\text{listrep}(p,l) \Rightarrow \text{iter}(p,i,l)) \]. \( \square \)


4.3 Summary

The examples above show that determining the soundness of magic-wand-involved assertion entailments can be very tricky. Especially when a proof rule is domain specific and its soundness is semantics specific, we have to be very careful with such proofs.

The following chapters propose the proof technique, magic-wand-as-frame. Besides the fact that it lets us write elegant proofs, this proof technique only uses those proof rules which derivable from the generic proof theory. This avoids the trouble from domain-specific and semantics-specific reasoning.
Chapter 5

Wand frame proof rules

In this chapter, I introduce wand-frame rules and two sets of variants: wandQ-frame rules and wandP-frame rules. I will also show some tiny applications of the magic-wand-as-frame approach. These examples are only used to show how magic-wand-as-frame works. Magic-wand-as-frame does NOT improve these proofs.

The case studies in Chapter 6 and 7 will show the power of magic-wand-as-frame.

5.1 Wand-frame rules

\[
\begin{align*}
\text{wand-frame-intro} & : \Sigma; \Gamma; Q \vdash P \rightarrow P * Q \\
\text{wand-frame-intro'} & : \Sigma; \Gamma; \text{emp} \vdash P \rightarrow P \\
\text{wand-frame-elim} & : \Sigma; \Gamma; P * (P \rightarrow Q) \vdash Q \\
\text{wand-frame-hor} & : \Sigma; \Gamma; (P_1 \rightarrow Q_1) * (P_2 \rightarrow Q_2) \vdash P_1 * P_2 \rightarrow Q_1 * Q_2 \\
\text{wand-frame-ver} & : \Sigma; \Gamma; (P \rightarrow Q) * (Q \rightarrow R) \vdash P \rightarrow R 
\end{align*}
\]

Only using these rules about magic wand means:

1. Never try to figure out internal structures of a heap which satisfies a wand assertion.

2. Only introduce a wand assertion by wand-frame-intro.
3. Only eliminate a wand assertion by \texttt{WAND-FRAME-ELIM}.

\textbf{WAND-FRAME} rules are all derivable from minimum propositional separation logic. Using them in verifications does not need extra soundness ensurance, or else those soundness proofs (as we have seen in Chapter 4) could be tedious and brittle.

Here is an example of using \texttt{WAND-FRAME} rules.

\begin{verbatim}
1 struct list {int head; struct list *tail;};
2 struct list * l;
3 int * p;
4
5 \{listrep([l], cons(x, z)) * \[p] \mapsto y\}
6 \{[l] \mapsto x * ([l] \mapsto y \leftarrow listrep([l], cons(y, z))) * \[p] \mapsto y\}
7 \↓ {[[l] \mapsto x * \[p] \mapsto y}\}
8       swap(&(l \rightarrow head), p);
9 \↑ {[[l] \mapsto y * \[p] \mapsto x}\}
10 \{[l] \mapsto y * \[p] \mapsto x * ([l] \mapsto y \leftarrow listrep([l], cons(y, z)))\}
11 \{listrep([l], cons(y, z)) * \[p] \mapsto x\}
\end{verbatim}

Lines 7-9 is a local verification of integer swapping: the C command transforms a program state satisfying \(([[l] \mapsto x * \[p] \mapsto y])\) to a state satisfying \(([[l] \mapsto y * \[p] \mapsto x])\). From those lines, we use \texttt{FRAME} to derive the triple of lines 6-10. The frame is \(([[l] \mapsto y \leftarrow listrep([l], cons(y, z)))\). We use \texttt{↓} and \texttt{↑} to highlight this usage of the frame rule. Line 6 introduces this frame by a transition from the precondition. Line 11 eliminates this magic wand and gets the postcondition. In this example, the local verification from line 7 to line 9 does not touch the wand conjunct at all. In other words, the wand conjunct is used as frame. This is why I call proof strategy \texttt{magic-wand-as-frame}. Since magic wands are only used in frames, one never needs to derive internal structural information from it. As we have shown in Chapter 4 such derivations can be very tricky.
Readers may have noticed that the wand frame in this example is a separating implication between a separating conjunct in the local postcondition and the corresponding conjunct in the global postcondition. Introducing a postcondition-related expression in the precondition (line 6) may be counterintuitive, but this is exactly how we should use magic wand.

Besides the introduction rule and the elimination rule, WAND-FRAME-HOR and WAND-FRAME-VER can be used for merging wand assertions, horizontally and vertically. We will see their applications later. In short, the vertical merging rule enables us to shrink the holes of partial data structures. The horizontal merging rule simply merges two holes into a larger one. The diagrams below illustrate these merging operations.

**Vertical composition:**

```
* ⊢ * ⊢ *
```

**Horizontal composition:**

```
* ⊢ * ⊢ *
```

### 5.2 WandQ-frame rules

Existential quantifiers are widely used in assertions when using separation logic to verify imperative programs’ functional correctness. For nonempty linked list listrep(p, cons(a, l)), unfolding it results in an existentially quantified assertion: \( \exists q. p \leftrightarrow a \ast p + 4 \leftrightarrow q \ast \text{listrep}(q, l) \). However, existential quantifiers affect the compositionality of wand-frames.

Using horizontal combination of wand-frames as an example here, shallow combination is sound but sometimes useless; in contrast, the rule for deep combination
would have been more useful but is unsound:

**SHALLOW-USELESS:**

\[ \Sigma; \Gamma; (\exists x. L_1(x) \rightarrow \exists x. G_1(x)) \ast (\exists y. L_2(y) \rightarrow \exists y. G_2(y)) \vdash \exists xy. L_1(x) \ast L_2(y) \rightarrow \exists xy. G_1(x) \ast G_2(y) \]

**DEEP-UNSOUND:**

\[ \Sigma; \Gamma; (\exists x. L_1(x) \rightarrow \exists x. G_1(x)) \ast (\exists x. L_2(x) \rightarrow \exists x. G_2(x)) \not\vdash \exists x. L_1(x) \ast L_2(x) \rightarrow \exists x. G_1(x) \ast G_2(x) \]

For example, from WAND-FRAME-INTRO, we know that,

\[ \Sigma; \Gamma; \quad p \mapsto a \ast p + 4 \mapsto q \vdash \text{listrep}(q, l) \ast \text{listrep}(p, \text{cons}(a, l)) \]

Sometimes, the local postcondition can only tell that \( l \) should satisfy some pure property \( A \) but cannot tell what the exact list \( l \) is. Then the following one can be useful:

\[ \Sigma; \Gamma; \quad p \mapsto a \ast p + 4 \mapsto q \vdash \exists l. A(l) \land \text{listrep}(q, l) \ast \exists l. A(l) \land \text{listrep}(p, \text{cons}(a, l)) \tag{5.1} \]

Now, let’s consider horizontal compositionality. We know that the following is sound:

\[ \Sigma; \Gamma; \quad p_1 \mapsto a_1 \ast p_1 + 4 \mapsto q_1 \ast p_2 \mapsto a_2 \ast p_2 + 4 \mapsto q_2 \vdash \exists l. A(l) \land \text{listrep}(q_1, l) \ast \text{listrep}(q_2, l) \ast \exists l. A(l) \land \text{listrep}(p_1, \text{cons}(a_1, l)) \ast \text{listrep}(p_2, \text{cons}(a_2, l)) \]

One can prove it by WAND-FRAME-INTRO but it cannot be proved combinatorially by \( (5.1) \) and SHALLOW-USELESS (or WAND-FRAME-HOR).
How can we use magic wand when local and global assertions are existentially quantified but gain deep compositionality? Use universal quantifier together with wands!

In the example above, instead of entailment (5.1), the following one is more useful:

\[
\Sigma; \Gamma; \quad p \mapsto a \cdot p + 4 \mapsto q \vdash \forall l. (A(l) \land \text{listrep}(q, l) \rightarrow A(l) \land \text{listrep}(p, \text{cons}(a, l)))
\]  \hspace{1cm} (5.2)

Why does it work? Because the witnesses of both existential quantifiers in (5.1) must be the same. Entailment (5.1) fails to reveal this property. Thus it is not strong enough for further combinational use. Actually, this is a very common property when we use existential quantifiers together with wands. Generally speaking, an entailment like (5.2) is stronger than (5.1) because:

\[
\Sigma; \Gamma; \quad \forall x. (L(x) \rightarrow G(x)) \vdash (\exists x. L(x)) \rightarrow (\exists x. G(x)).
\]

The following entailments show the proof rules for using wands together with quantifiers. \textsc{wandQ-frame-ver} and \textsc{wandQ-frame-hor} show the compositionality of quantified wands. Here, “Q” represents quantifiers.

\textsc{wandQ-frame-intro:} \hspace{0.5cm} \Sigma; \Gamma; Q \vdash \forall x. (P(x) \rightarrow P(x) \rightarrow Q) \\
\textsc{wandQ-frame-intro’:} \hspace{0.5cm} \Sigma; \Gamma; \text{emp} \vdash \forall x. (P(x) \rightarrow P(x)) \\
\textsc{wandQ-frame-elim:} \hspace{0.5cm} \Sigma; \Gamma; P(x_0) \cdot \forall x. (P(x) \rightarrow Q(x)) \vdash Q(x_0) \\
\textsc{wandQ-frame-hor:} \hspace{0.5cm} \Sigma; \Gamma; \forall x. (P_1(x) \rightarrow Q_1(x)) \cdot \forall x. (P_2(x) \rightarrow Q_2(x)) \\
\hspace{2cm} \vdash \forall x. (P_1(x) \cdot P_2(x) \rightarrow Q_1(x) \cdot Q_2(x)) \\
\textsc{wandQ-frame-ver:} \hspace{0.5cm} \Sigma; \Gamma; \forall x. (P(x) \rightarrow Q(x)) \cdot \forall x. (Q(x) \rightarrow R(x)) \\
\hspace{2cm} \vdash \forall x. (P(x) \rightarrow R(x)) \\
\textsc{wandQ-frame-refine:} \hspace{0.5cm} \Sigma; \Gamma; \forall x. (P(x) \rightarrow Q(x)) \vdash \forall y. (P(f(y)) \rightarrow Q(f(y)))
5.3 WandP-frame rules

One obstacle of using magic wand as frame is the side condition of the FRAME rule.

\[
\text{FRAME: } \begin{cases}
\Sigma; \Gamma \vdash \{ P \} c \left\{ Q, [\vec{R}] \right\} \text{ and } \\
\text{any nonaddressable variable which is modified} \\
in c \text{ does not freely occur in } F \\
\text{then } \Sigma; \Gamma \vdash \{ P \ast F \} c \left\{ Q \ast F, [\vec{R} \ast F] \right\}.
\end{cases}
\]

It becomes tricky to apply WAND-FRAME method in situations that some local variables are modified. For example, how can we derive (5.4) from (5.3)?

\[
\begin{align*}
\{[p] \mapsto h \} \times = p \rightarrow \text{head}; \{[p] \mapsto [x]\} & \quad (5.3) \\
\{\text{listrep}([p], \text{cons}(h,t))) \times = p \rightarrow \text{head}; \{\text{listrep}([p], \text{cons}([x], t))\} & \quad (5.4)
\end{align*}
\]

We would like to use a wand-frame like

\[ [p] \mapsto [x] \ast \text{listrep}([p], \text{cons}([x], t)). \]

The problem is that the modified program variable \( x \) freely occurs in the wand-frame, which conflicts with the side conditions of FRAME. In order to make wand-frame interact well with such assertions (that program variable denotations are mixed into other assertions), we need a way to abstract over current values of program variables.

To handle this situation, I introduce a new operator for quantifying over “possible values of modified variables”. Given a sequence of program variables \( \vec{x} \) and a separation logic assertion \( P \), I use \( [\vec{x}]P \) to express: no matter how the variables in \( \vec{x} \) are modified, the assertion \( P \) still holds.
Formally, $[\vec{x}]P$ represents the abbreviation of $\forall \vec{x}. P[\vec{x}/\vec{x}]$. For example,

$$
[x]P \triangleq \forall x. P[x/x]
$$

$$
[x;y]P \triangleq \forall x,y. P[x/x; y/y]
$$

I borrow the notation $[\cdot]$ from dynamic logics [23]. On one hand, the semantic meaning of $[\vec{x}]P$ is an implicit universal quantification over a set of possible program state transformations—replacing the denotations of $\vec{x}$ with $\vec{x}$. On the other hand, $[\vec{x}]P$ satisfies some basic proof rules of the necessity modality (dynamic logics are a special kind of modal logic extensions). The following two rules are sound:

$$
\Sigma; \Gamma; [\vec{x}]P \vdash P \quad (5.5)
$$

If $\Sigma; \Gamma; P \vdash Q$ then $\Sigma; \Gamma; [\vec{x}]P \vdash [\vec{x}]Q \quad (5.6)$

Also, this modality partially commutes with separating conjunction:

$$
\Sigma; \Gamma; [\vec{x}]P \ast [\vec{x}]Q \vdash [\vec{x}](P \ast Q) \quad (5.7)
$$

The following rule describes the behavior of $[\vec{x}]P$ in special cases:

$$
\text{If none of } \vec{x} \text{ freely occur in } P, \text{ then } \Sigma; \Gamma; P \vdash [\vec{x}]P \quad (5.8)
$$

All these four rules can be derived from minimum first order separation logic.

Although the rules above for $[\vec{x}]P$ are elegant, I propose to only use this modality in wand-frames via the following WANDP-FRAME rules. They are all derivable from minimum propositional separation logic logic and (5.5) (5.6) (5.7) (5.8). Here, “P” represents program variables.
wandP-frame-intro: If none of $\vec{x}$ freely occur in $Q$ then $\Sigma; \Gamma; Q \vdash [\vec{x}](P \rightarrow P \ast Q)$

wandP-frame-intro': $\Sigma; \Gamma; \text{emp} \vdash [\vec{x}](P \rightarrow P)$

wandP-frame-elim: $\Sigma; \Gamma; P \ast [\vec{x}](P \rightarrow Q) \vdash Q$

wandP-frame-hor: $\Sigma; \Gamma; [\vec{x}](P_1 \rightarrow Q_1) \ast [\vec{x}](P_2 \rightarrow Q_2) \vdash [\vec{x}](P_1 \ast P_2 \rightarrow Q_1 \ast Q_2)$

wandP-frame-ver: $\Sigma; \Gamma; [\vec{x}](P \rightarrow Q) \ast [\vec{x}](Q \rightarrow R) \vdash [\vec{x}](P \rightarrow R)$

Now, let’s go back to the problem at the beginning of this section. Using our new modality, we have a solution there: $[x] \ (\mathbb{J} \ p \ K \mapsto \rightarrow \mathbb{J} x K \rightarrow P \ast \mathbb{J} p K \mapsto \rightarrow \mathbb{J} x K \rightarrow \text{cons}(\mathbb{J} p K, t)))$ can be used as frame. Here is the proof.

1. $\{\text{listrep}(\mathbb{J} p K, \text{cons}(h, t))\}$
2. $\{\mathbb{J} p K \mapsto h \ast \exists q. \ [\mathbb{J} p K] \mapsto q \ast \text{listrep}(q, t)\}$
3. $\{\mathbb{J} p K \mapsto h \ast x \ (\mathbb{J} p K \mapsto x) \rightarrow \mathbb{J} p K \mapsto x \ast \exists q. \ [\mathbb{J} p K] \mapsto q \ast \text{listrep}(q, t)\}$
4. $\{\mathbb{J} p K \mapsto h \ast x \ (\mathbb{J} p K \mapsto x) \rightarrow \text{listrep}(\mathbb{J} p K, \text{cons}(x, t))\}$
5. $\n \n x = p \rightarrow \text{head};$
6. $\{\mathbb{J} p K \mapsto x\}$
7. $\{\mathbb{J} p K \mapsto x \ast x \ (\mathbb{J} p K \mapsto x) \rightarrow \text{listrep}(\mathbb{J} p K, \text{cons}(x, t))\}$
8. $\{\text{listrep}(\mathbb{J} p K, \text{cons}(x, t))\}$

The transition from line 2 to line 3 is by wandP-frame-intro. The transition from line 8 to line 9 is due to wandP-frame-elim.

The application of frame is legal now because $x$ does not freely occur in

$[x] \ (\mathbb{J} p K \mapsto x \ast \text{listrep}(\mathbb{J} p K, \text{cons}(x, t))).$

Generally speaking, none of $\vec{x}$ freely occur in $[\vec{x}]P$. 

51
Chapter 6

Case study: linked list append

In this chapter, I use magic wand as frame to verify the following C program.

```c
struct list {int head; struct list *tail;};
struct list * append1 (struct list * x, struct list * y) {
    struct list * t, * u;
    if (x == NULL)
        return y;
    else {
        t = x; u = t -> tail;
        while (u != NULL) {
            t = u; u = t -> tail;
        }
        t -> tail = y;
        return x;
    }
}
```
The verification goal is to show: for any $l_1$ and $l_2$,

$$\{ \text{listrep}([x], l_1) \ast \text{listrep}([y], l_2) \}$$

$\text{append1}(x, y)$

$$\{ \text{listrep}([\text{ret_temp}], l_1 \cdot l_2) \}$$

In the postcondition, $[\text{ret_temp}]$ represents the return value.

### 6.1 Wand frame proof

The first proof that I demonstrate here is to use WAND-FRAME rules.

```plaintext
1 \ \{ \text{listrep}([x], l_1) \ast \text{listrep}([y], l_2) \} \\
2 \ \{ \exists x, y. \ [x] = x \land [y] = y \land \text{listrep}(x, l_1) \ast \text{listrep}(y, l_2) \} \\
3 \ \{ \forall x \text{ and } y \} \\
4 \ \{ [x] = x \land [y] = y \land \text{listrep}(x, l_1) \ast \text{listrep}(y, l_2) \} \\
5 \ \text{if } (x == \text{NULL}) \\
6 \ \{ [y] = y \land l_1 = \text{nil} \land \text{listrep}(y, l_2) \} \\
7 \ \text{return } y; \\
8 \ \text{else} \{ \\
9 \ \{ \exists a, l_1', p. \ [x] = x \land [y] = y \land l_1 = \text{cons}(a, l_1') \land \\
10 \ \{ x \mapsto h \ast x + 4 \mapsto p \ast \text{listrep}(p, l_1') \ast \text{listrep}(y, l_2) \} \\
11 \ \} \\
12 \ \{ \exists a, l_1', p. \ [x] = x \land [y] = y \land [t] = x \land [u] = p \land l_1 = \text{cons}(a, l_1') \land \\
13 \ \{ x \mapsto a \ast x + 4 \mapsto p \ast \text{listrep}(p, l_1') \ast \text{listrep}(y, l_2) \} \\
14 \ \} \\
15 \ \text{Let } \text{Frame}(t, l) \triangleq \text{listrep}(t, l \cdot l_2) \ast \text{listrep}(x, l_1 \cdot l_2) \\
```
\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = u \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l_1') \ast \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, l_1')) \]

while \((u \neq \text{NULL})\) {

\[ \Backslash \Backslash \{ \ \\
\] Given \(a, l_1', t, u\).

\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = u \land \]
\[ u \neq \text{null} \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l_1') \ast \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, l_1')) \]

\[ \Backslash \Backslash \{ \ \\
\]

\[ \exists b, l_1'', p. \ [x] = x \land [y] = y \land [t] = t \land [u] = u \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l_1'') \ast \\
\]
\[ \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, \text{cons}(b, l_1''))) \]

\[ \Backslash \Backslash \{ \ \\
\]

Given \(b, l_1''\) and \(p\).

\[ \exists b, l_1'', p. \ [x] = x \land [y] = y \land [t] = t \land [u] = u \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l_1'') \ast \\
\]
\[ \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, \text{cons}(b, l_1''))) \]

\[ t = u; \ u = t \mapsto \text{tail}; \]

\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = p \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l_1'') \ast \\
\]
\[ \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, \text{cons}(b, l_1''))) \]

\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = p \land \\
\]
\[ u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l_1'') \ast \\
\]
\[ \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, \text{cons}(b, l_1''))) \]

\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = p \land \\
\]
\[ u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l_1'') \ast \text{listrep}(y, l_2) \ast \\
\]
\[ \text{Frame}(u, \text{cons}(b, l_1'')) \]

\[ \exists a, l_1', t, u. \ [x] = x \land [y] = y \land [t] = t \land [u] = u \land \\
\]
\[ t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l_1') \ast \text{listrep}(y, l_2) \ast \\
\]
\[ \text{Frame}(t, \text{cons}(a, l_1')) \]
Given $a, l_1', t$ and $u$.

\[
\begin{aligned}
[x] &= x \land [y] = y \land [t] = t \land [u] = u \land u = \text{null} \\
\end{aligned}
\]

\[
\begin{aligned}
t \mapsto a \ast t + 4 \mapsto & u \ast \text{listrep}(u, l_1') \ast \text{listrep}(y, l_2) \ast \\
\text{Frame}(t, \text{cons}(a, l_1'))
\end{aligned}
\]

\[
\begin{aligned}
[x] &= x \land [y] = y \land [t] = t \\
\end{aligned}
\]

\[
\begin{aligned}
t \mapsto a \ast t + 4 \mapsto & \text{null} \ast \text{listrep}(y, l_2) \ast \text{Frame}(t, [a])
\end{aligned}
\]

\[
t \mapsto \text{tail} = y;
\]

\[
\begin{aligned}
\{ [x] = x \land t \mapsto a \ast t + 4 \mapsto y \ast \text{listrep}(y, l_2) \ast \text{Frame}(t, [a]) \}
\end{aligned}
\]

\[
\begin{aligned}
\{ [x] = x \land \text{listrep}(t, \text{cons}(a, l_2)) \ast \text{Frame}(t, [a]) \}
\end{aligned}
\]

\[
\begin{aligned}
\{ [x] = x \land \text{listrep}(x, l_1 \cdot l_2) \}
\end{aligned}
\]

\[
\text{return } x;
\]

The most important part of this proof is the while loop. I use the following loop invariant:

\[
\exists a, l_1', t, u. [x] = x \land [y] = y \land [t] = t \land [u] = u \land t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l_1') \ast \text{listrep}(y, l_2) \ast \text{Frame}(t, \text{cons}(a, l_1'))
\]

in which \(\text{Frame}(t, l) \triangleq \text{listrep}(t, l \cdot l_2) \rightarrow \text{listrep}(x, l_1 \cdot l_2)\). See also line 12 and 13.

The transition from line 11 to line 13 shows that the loop’s precondition implies the loop invariant. Using \textsc{wand-frame-intro’}, we know:

\[
l_1; l_2; x; y; a; l_1'; p \quad \text{emp} \vdash \text{listrep}(x, l_1 \cdot l_2) \rightarrow \text{listrep}(x, l_1 \cdot l_2)
\]

Since we know $l_1 = \text{cons}(a, l_1')$, the right side of this entailment is just \(\text{Frame}(x, \text{cons}(a, l_1'))\). Thus, it proves the validity of this transition from line 11 to line 13 by instantiating the existentially quantified variable $t$ with $x$. 

55
Lines \([16, 24]\) verify the loop body. We first know \(u \neq \text{null}\) from the loop condition (see line \([16]\)). It tells us that the linked list starting from it cannot be empty, which justify the transition to line \([19]\). Line \([21]\) is the strongest postcondition of executing the loop body from the precondition in line \([19]\). The transition from line \([21]\) to \([22]\) is an application of \textsc{wand-frame-intro}. The transition from line \([22]\) to \([23]\) is an application of \textsc{wand-frame-ver}. The loop invariant is reestablished at line \([24]\).

Specifically, the existentially quantified variables \(t\) and \(u\) are instantiated with \(u\) and \(p\).

After the loop, since we know \(u = \text{null}\) (see line \([27]\)), the linked list stored there must be nil (see line \([28]\)). The C command “\(t \rightarrow \text{tail} = y\)” finally appends the second linked list to the tail of first one. The transition from line \([31]\) to line \([32]\) in the end is interesting—it is actually an instance of \textsc{wand-frame-elim}:

\[
l_1; l_2; x; y; a; t; \quad \text{listrep}(t, \text{cons}(a, l_2)) \ast (\text{listrep}(t, \text{cons}(a, l_2)) \rightarrow \text{listrep}(x, l_1 \cdot l_2)) \vdash \text{listrep}(x, l_1 \cdot l_2)
\]

The second separating conjunct on the left side is exactly \(\text{Frame}(t, [a])\).

In short, I establish an empty wand frame first then merge a single layer wand frame into it in every execution of loop body. After the loop, this wand frame (with quantifier) is eliminated.

### 6.2 WandQ frame proof

The second proof uses \textsc{wandQ-frame} rules.

First of all, we can define list segments by a combination of magic wand and universal quantifier:

\[
lseg(p, q, l) \triangleq \forall l'. (\text{listrep}(q, l') \ast \text{listrep}(p, l \cdot l'))
\]
Based on \textsc{wandQ-frame} rules, we get the following properties of list segments:

\[
\begin{align*}
\Sigma; \Gamma; \text{emp} & \vdash \text{lseg}(p, q, \text{nil}) \quad \text{(6.1a)} \\
\Sigma; \Gamma; p & \mapsto a \ast p + 4 \mapsto q \vdash \text{lseg}(p, q, [a]) \quad \text{(6.1b)} \\
\Sigma; \Gamma; \text{lseg}(p, q, l_1) \ast \text{listrep}(q, l_2) & \vdash \text{listrep}(p, l_1 \cdot l_2) \quad \text{(6.1c)} \\
\Sigma; \Gamma; \text{lseg}(p, q, l_1) \ast \text{lseg}(q, r, l_2) & \vdash \text{lseg}(p, r, l_1 \cdot l_2) \quad \text{(6.1d)}
\end{align*}
\]

Specifically, \textsc{wandQ-frame-intro’} proves (6.1a). \textsc{wandQ-frame-intro} proves (6.1b). Entailment (6.1c) can be proved \textsc{wandQ-frame-elim}. The proof of (6.1d) is worth more lines here:

\[
\begin{align*}
\text{lseg}(p, q, l_1) \ast \text{lseg}(q, r, l_2) \\
= \forall l'. (\text{listrep}(q, l') \rightarrow \text{listrep}(p, l_1 \cdot l')) \ast \\
\forall l''. (\text{listrep}(r, l'') \rightarrow \text{listrep}(q, l_2 \cdot l'')) \\
\vdash \forall l''. (\text{listrep}(q, l_2 \cdot l'') \rightarrow \text{listrep}(p, l_1 \cdot l_2 \cdot l'')) \ast \\
\forall l''. (\text{listrep}(r, l'') \rightarrow \text{listrep}(q, l_2 \cdot l'')) \\
\vdash \forall l''. (\text{listrep}(r, l'') \rightarrow \text{listrep}(p, l_1 \cdot l_2 \cdot l'')) \\
= \text{lseg}(p, r, l_1 \cdot l_2)
\end{align*}
\]

The first entailment is due to \textsc{wandQ-frame-refine} and the second is an instance of \textsc{wandQ-frame-nerve}.

Here is the program verification using this nonrecursively defined list segment predicate.
\[ \{ \text{listrep}(\[x\], l_1) \ast \text{listrep}(\[y\], l_2) \} \]
\[ \{ \exists x, y. \[x\] = x \land \[y\] = y \land \text{listrep}(x, l_1) \ast \text{listrep}(y, l_2) \} \]
\[ \{ \text{Given } x \text{ and } y \} \]
\[ \{ \[x\] = x \land \[y\] = y \land \text{listrep}(x, l_1) \ast \text{listrep}(y, l_2) \} \]
\[ \text{if } (x == \text{NULL}) \]
\[ \{ \[y\] = y \land l_1 = \text{nil} \land \text{listrep}(y, l_2) \} \]
\[ \text{return } y; \]
\[ \text{else } \{ \]
\[ \{ \exists a, l', p. \[x\] = x \land \[y\] = y \land l_1 = \text{cons}(a, l') \land \
\{ x \mapsto h \ast x + 4 \mapsto p \ast \text{listrep}(p, l') \ast \text{listrep}(y, l_2) \} \]
\[ t = x; \ u = t \mapsto \text{tail}; \]
\[ \{ \exists a, l', l_1', t, u. \[x\] = x \land \[y\] = y \land \[t\] = x \land \[u\] = p \land l_1 = \text{cons}(a, l_1') \land \
\{ x \mapsto a \ast x + 4 \mapsto p \ast \text{listrep}(p, l_1') \ast \text{listrep}(y, l_2) \} \}
\[ \{ \exists b, l''_1, p. \[x\] = x \land \[y\] = y \land \[t\] = t \land \[u\] = u \land l_1 = \text{cons}(b, l''_1) \land \
\{ t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l_1') \} \}
\[ \text{while } (u != \text{NULL}) \} \]
\[ \{ \text{Given } a, l'_1, l''_1, t \text{ and } u. \text{ Assume } l = l'_1 \ast \text{cons}(a, l''_1). \}
\[ \{ \exists a, t + 4 \mapsto u \ast \text{listrep}(u, l''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l_1') \} \]
\[ \{ \exists b, l''_1, p. \[x\] = x \land \[y\] = y \land \[t\] = t \land \[u\] = u \land l_1 = \text{cons}(b, l''_1) \land \
\{ t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l_1') \} \}
\[ \{ \text{Given } b, l''_1 \text{ and } p. \text{ Assume } l''_1 = \text{cons}(b, l''_1). \}
\]

58
\[
\begin{align*}
\{ [x] &= x \land [y] = y \land [t] = t \land [u] = u \land \\
t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \\
\text{listrep}(p, l'''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) \}
\end{align*}
\]

\[
\{ [x] = x \land [y] = y \land [t] = u \land [u] = p \land \\
t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \\
\text{listrep}(p, l'''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) \}
\]

\[
\begin{align*}
\{ [x] &= x \land [y] = y \land [t] = u \land [u] = p \land \\
u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l'''_1) \ast \\
\text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) \}
\end{align*}
\]

\[
\begin{align*}
\exists a, l'_1, l'''_1, t, u. \ [x] &= x \land [y] = y \land [t] = t \land [u] = u \land \\
\text{listrep}(u, l'''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]

\[
\begin{align*}
\{ [x] &= x \land [y] = y \land [t] = t \land [u] = u \land \text{null} \land \\
t \mapsto a \ast t + 4 \mapsto u \ast \text{listrep}(u, l'''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]

\[
\begin{align*}
\{ [x] &= x \land [y] = y \land [t] = t \land [u] = u \land l'''_1 = \text{nil} \land \\
t \mapsto a \ast t + 4 \mapsto \text{null} \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]

Assume \( l''_1 = \text{nil} \).

\[
\begin{align*}
\{ [x] &= x \land [y] = y \land [t] = u \land \\
t \mapsto a \ast t + 4 \mapsto \text{null} \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]

Assume \( l''_1 = \text{nil} \).

\[
\begin{align*}
\{ [x] &= x \land t \mapsto a \ast t + 4 \mapsto y \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]

\[
\begin{align*}
\{ [x] &= x \land \text{listrep}(t, \text{cons}(a, l_2)) \ast \text{lseg}(x, t, l'_1) 
\end{align*}
\]
Similarly, an empty wand frame is established at line 12 by rule (6.1a). A single layer wand frame is established at line 21 by rule (6.1b) and gets merged into the major wand frame at line 22 by rule (6.1d). After the loop, the wand frame is eliminated at line 33 by rule (6.1c). Recall that these four rules about list segment are exactly instances of quantifier wand frame rules!

### 6.3 Another wandQ frame proof

In two previous proofs, I isolate program variables from spatial separating conjuncts by using extra logic variables. For example, listrep([x], l_1) are rewritten as \( \exists x. [x] = x \land \text{listrep}(x, l_1) \).

After extracting existentials into the context, modified nonaddressable program variables do not freely occur in separating conjuncts of preconditions, especially those wand-frame conjuncts. As a result, we can legally apply the \textsc{frame} rule, explicitly or implicitly. For example, in section 6.2 the Hoare triple from line 18 to 20 applies the \textsc{frame} rule:

\[
\begin{array}{ll}
& \{ [x] = x \land [y] = y \land [t] = t \land [u] = u \} \\
\downarrow & t \mapsto a \ast t + 4 \mapsto u \ast u \mapsto b \ast u + 4 \mapsto p \ast \\
& \text{listrep}(p, l''_1) \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l'_1) \\
\downarrow & \{ [t] = t \land [u] = u \land u \mapsto b \ast u + 4 \mapsto p \}
\end{array}
\]

\[
\begin{array}{ll}
& t = u; u = t \mapsto \text{tail}; \\
\downarrow & \{ [t] = u \land [u] = p \land u \mapsto b \ast u + 4 \mapsto p \}
\end{array}
\]
In comparison, it is conventional in Hoare logic and separation logic to mix the denotations (current values) of program variables into other assertions. For example, the loop invariant above can be written as:

\[
\begin{align*}
&
\{ [t] \mapsto a \cdot [t] + 4 \mapsto [u] \mapsto b \cdot [u] + 4 \mapsto p \} \\
&\quad \text{listrep}(p, [t]') \ast \text{listrep}(y, l_2) \ast \text{lseg}(x, t, l_1')
\end{align*}
\]

In the postcondition, the existentially quantified \( t' \) represents the old value of \( t \).

Both proof styles have their own pros and cons. The conventional way can make assertions shorter. The isolated-program-variable way enables us to apply the \textit{frame} rule more conveniently. I will use this isolated-program-variable style in later case studies and revisit it in Part II where this proof style plays a more important role for proof automation.

In this section, I show another \textit{wandQ-frame} proof for linked list append. This proof shows that magic-wand-as-frame is also compatible with the conventional assertion style too.

```plaintext
1 \{ \text{listrep}([x], l_1) \ast \text{listrep}([y], l_2) \}
2 \text{if} (x == \text{NULL})
3 \{ l_1 = \text{nil} \land \text{listrep}([y], l_2) \}
4 \quad \text{return} y;
5 \text{else} 
6 \{ \exists a, l'_1. l_1 = \text{cons}(a, l'_1) \land \text{listrep}([x], \text{cons}(a, l'_1)) \ast \text{listrep}([y], l_2) \}
7 \quad t = x; u = t \rightarrow \text{tail};
```
\begin{align*}
&\{\exists a, l'_1. \ l_1 = \text{cons}(a, l'_1) \land [x] = [t] \land \\
&\ [t] \mapsto a \ast [t] + 4 \mapsto [u] \ast \text{listrep}([u], l'_1) \ast \text{listrep}([y], l_2)\} \\
&\{\exists a, l'_1, l'''_1. l_1 = l'_1 \cdot \text{cons}(a, l'''_1) \land [t] \mapsto a \ast [t] + 4 \mapsto [u] \ast \\
&\text{listrep}([u], l'''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1)\} \\
&\text{while } (u \neq \text{NULL}) \{ \\
&\{\text{Given } a, l'_1, l'''_1. \text{ Assume } l = l'_1 \cdot \text{cons}(a, l'''_1). \\
&\{[u] \neq \text{null } \land [t] \mapsto a \ast [t] + 4 \mapsto [u] \ast \\
&\text{listrep}([u], l'''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1)\} \\
&\{\exists b, l'''_1, p. l'''_1 = \text{cons}(b, l'''_1) \land [t] \mapsto a \ast [t] + 4 \mapsto [u] \ast \\
&\text{listrep}([u], \text{cons}(b, l'''_1)) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1)\} \\
&t = u; u = t \to \text{tail}; \\
&\{\exists l''. l'' \mapsto a \ast l'' + 4 \mapsto [t] \ast [t] \mapsto b \ast [t] + 4 \mapsto [u] \}
&\text{listrep}([u], l'''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], l'', l'_1) \\
&\{\exists l''. \text{lseg}(l'', [t], [a]) \ast [t] \mapsto b \ast [t] + 4 \mapsto [u] \}
&\text{listrep}([u], l'''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], l'', l'_1) \\
&\{[t] \mapsto b \ast [t] + 4 \mapsto [u] \}
&\text{listrep}([u], l'''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1) \\
&\{\exists a, l'_1, l''_1. l_1 = l'_1 \cdot \text{cons}(a, l''_1) \land [t] \mapsto a \ast [t] + 4 \mapsto [u] \ast \\
&\text{listrep}([u], l''_1) \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1)\} \\
&\{l''_1 = \text{nil } \land [t] \mapsto a \ast [t] + 4 \mapsto \text{null } \ast \text{listrep}([y], l_2) \ast \text{lseg}([x], [t], l'_1)\} \\
&\text{Assume } l'''_1 = \text{nil}. \\
\end{align*}
\{ t \mapsto \text{tail} = y; \}

\{ t \mapsto \text{tail} = y; \}

\{ \text{listrep}(t, \text{cons}(a, l)) \mapsto \text{lseg}(x, t, l'_1) \}

\{ \text{listrep}(x, l'_1 \cdot \text{cons}(a, l)) \}

\{ \text{listrep}(x, l_1 \cdot l_2) \}

\{ \text{listrep}(x, \text{cons}(a, l)) \mapsto \text{lseg}(x, t, l'_1) \}

\{ \text{listrep}(x, l_1 \cdot l_2) \}

6.4 Comparison with recursive predicates

I am not the first one to verify programs with linked lists. Most previous work would use a recursive predicate, for example \text{lseg\_rec}_{\cdot \cdot \cdot} to complete the functional correctness proof for linked list append. Recall that it is defined recursively so that:

\begin{align*}
\Sigma; \Gamma; \text{lseg\_rec}(p, q, \text{nil}) & \vdash p = q \land \text{emp} \\
\Sigma; \Gamma; \text{lseg\_rec}(p, q, \text{cons}(a, l)) & \vdash \\
& \exists p'. p \mapsto a \ast p + 4 \mapsto p' \ast \text{lseg\_rec}(p', q, l)
\end{align*}

With this alternative predicate, counterparts of (6.1a), (6.1b), (6.1c) and (6.1d) are still sound:

\begin{align*}
\Sigma; \Gamma; \text{emp} & \vdash \text{lseg\_rec}(p, q, \text{nil}) \quad (6.2) \\
\Sigma; \Gamma; p \mapsto a \ast p + 4 \mapsto q & \vdash \text{lseg\_rec}(p, q, [a]) \quad (6.3) \\
\Sigma; \Gamma; \text{lseg\_rec}(p, q, l_1) \ast \text{listrep}(q, l_2) & \vdash \text{listrep}(p, l_1 \cdot l_2) \quad (6.4) \\
\Sigma; \Gamma; \text{lseg\_rec}(p, q, l_1) \ast \text{lseg\_rec}(q, r, l_2) & \vdash \text{lseg\_rec}(p, r, l_1 \cdot l_2) \quad (6.5)
\end{align*}

With these proof rules, the proofs in section 6.2 and section 6.3 are still valid.
However, using this kind of recursive predicates for list segments has a problem: the soundness of its proof rules are actually very brittle. Different authors had proposed different recursive predicates for list segments. The following one, \texttt{nt\_lseg\_rec}, called “no-touch list segment” \cite{9}, is used by separation logic verification tools like Smallfoot’s \cite{2}.

\[
\Sigma; \Gamma; \text{nt\_lseg\_rec}(p, q, \text{nil}) \vdash p = q \land \text{emp}
\]

\[
\Sigma; \Gamma; \text{nt\_lseg\_rec}(p, q, \text{cons}(a, l)) \vdash
\exists p'. p \neq r \land p \mapsto \to a \ast p + 4 \mapsto \to p' \ast \text{nt\_lseg\_rec}(p', q, l)
\]

These two recursive predicates look similar, but their proof theories are surprisingly different. For example, the counterpart of (6.1d) (or (6.5)) is unsound:

\[
\Sigma; \Gamma; \text{lt\_lseg\_rec}(p, q, l_1) \ast \text{nt\_lseg\_rec}(q, r, l_2) \not\vdash \text{nt\_lseg\_rec}(p, r, l_1 \cdot l_2)
\]

The no-touch recursive list segment predicate only satisfies a weaker rule:

\[
\Sigma; \Gamma; \text{nt\_lseg\_rec}(p, q, l_1) \ast \text{nt\_lseg\_rec}(q, r, l_2) \ast \text{nt\_lseg\_rec}(r, s, l_3) \vdash
\text{nt\_lseg\_rec}(p, r, l_1 \cdot l_2) \ast \text{nt\_lseg\_rec}(r, s, l_3)
\]

In short, recursive predicates are stronger predicates and we can reveal internal structural information from them. For example,

\[
\Sigma; \Gamma; \text{lseg\_rec}(p, r, l_1 \cdot l_2) \vdash \exists q. \text{lseg\_rec}(p, q, l_1) \ast \text{lseg\_rec}(q, r, l_2)
\]

\[
\Sigma; \Gamma; \text{lseg}(p, r, l_1 \cdot l_2) \not\vdash \exists q. \text{lseg}(p, q, l_1) \ast \text{lseg}(q, r, l_2)
\]

But magic wand expressions are more convenient for merging segments together. The soundness of recursive list segments’ merging rules are brittle and need inductive
justification. (Those proof rules can still be sound. But a small modification in
the recursive definition may break them.) The soundness of wand list segments are
derivable from minimum separation logic.
Chapter 7

Case study: binary search tree insert

In this chapter, I verify a C implementation of binary search tree (BST) insert using magic-wand-as-frame.

Figure 7.1 shows this insert function. It does not return a new tree, but rather modifies the old tree. Since even the root address can be modified, the argument of this function has type pointer-to-pointer-to-tree, which we call treebox.

Consider running insert(p0,8,"h"), where p0 points to a treebox containing the root of a tree as shown in Figure 7.2. After one iteration of the loop or two iterations, variable p contains address p, which is a treebox containing a pointer to a subtree. We use the C type treebox here again because this subtree and its root pointer may be modified in the future loop iterations.

Remark: Linked list append and BST insert look similar—both of them traverse a data structure from the root and make some modification internally. Thus I intentionally choose these two different style of C implementation in order to show the flexibility of magic-wand-as-frame. (1) In the linked list example, the loop is only used to traverse and reach the tail point. The modification step is done after the loop. In
struct tree {
    int key;
    void *value;
    struct tree *left, *right;
};

typedef struct tree **treebox;

void insert (treebox p, int x, void *v) {
    struct tree *q;
    while (1) {
        q = *p;
        if (q==NULL) {
            q = (struct tree *) surely_malloc (sizeof *p);
            q->key=x; q->value=v;
            q->left=NULL; q->right=NULL;
            *p=q;
            return;
        } else {
            int y = q->key;
            if (x<y) {
                p= &q->left;
            } else if (y<x) {
                p= &q->right;
            } else {
                q->value=v;
                return;
            }
        }
    }
}

Figure 7.1: Binary Search Tree insertion

Figure 7.2: Execution of insert(p0, 8, "h").

contrast, the modification is completed inside the loop in the BST example. (2) In the linked list example, both current vertex u and parent vertex t are recorded. When u reaches null, the modification will happen at address p→next. In contrast, only a pointer to the current, p, vertex is recorded in BST insert. When p points to null, the new allocated vertex is directly assigned to *p. This is natural and reasonable, because it is not enough to record the parent vertex—an extra bit for left-or-right is needed.

In program verification, it is important to distinguish an algorithm and its implementation. In this case, the BST insert algorithm describes how to compute the
insertion result, given a BST, a key and a value as input. As shown in Figure 7.3, BST is a set of inductively defined math objects and the math function \( \text{ins} \) (as our description of the algorithm) computes the result recursively. On the other side, its C implementation fills in the details like how a BST is stored in memory and how to complete the computation steps by operations on memories and variables.

Binary trees:
\[
t ::= \ N \mid T(t_1, k, v, t_2)
\]

Math function as algorithm description:
\[
\text{ins}(N, x, v) \triangleq T(N, x, v, N)
\]
\[
\text{ins}(T(t_1, x_0, v_0, t_2), x, v) \triangleq \\
\begin{cases}
\text{If } x < x_0, & T(\text{ins}(t_1, x, v), x_0, v_0, t_2) \\
\text{If } x = x_0, & T(t_1, x, v, t_2) \\
\text{If } x > x_0, & T(t_1, x_0, v_0, \text{ins}(t_2, x, v))
\end{cases}
\]

Figure 7.3: Algorithm Description

The program correctness proof can be divided into two parts: algorithm correctness and implementation correctness. We know that a binary search tree should always preserve the binary-search-tree property, i.e. keys are in order. In this example, algorithm correctness says: if the input binary tree satisfies the binary-search-tree property, then (1) the output of this ins function has that property as well; (2) the tree implements a finite mapping, i.e. the key-value mapping in the output tree corresponds to the usual update result of a finite map. The implementation correctness says that this C \text{insert} function correctly implements the ins algorithm of BST insert.

The implementation-correctness proof is independent of and does not require the binary-search-tree property. This illustrates a \textit{modular} proof style. When algorithm correctness is trivial, its proof can be combined together with the implementation-correctness proof—linked list append is such an example. Correctness of BST operations is nontrivial, therefore modularity is important. Thus I choose to separate these
two proof components. In this chapter, I will verify the implementation correctness of BST insert.

## 7.1 Separation logic specification of implementation correctness

It is routine in separation logic verification to define a representation predicate for specific data structures.

**Recursive Properties for Tree-Box Representation:**

\[
\Sigma; \Gamma; \text{TBrep}(p, N) \not\vdash p \rightarrow null
\]

\[
\Sigma; \Gamma; \text{TBrep}(p, T(t_1, k, v, t_2)) \not\vdash \exists q. p \rightarrow q \ast
\]

\[
q.\text{key} \mapsto k \ast q.\text{value} \mapsto v \ast \text{TBrep}(q, t_1) \ast \text{TBrep}(q, t_2)
\]

**Recursive Properties for Tree Representation:**

\[
\Sigma; \Gamma; \text{Trep}(p, N) \not\vdash p = null \land \text{emp}
\]

\[
\Sigma; \Gamma; \text{Trep}(p, T(t_1, k, v, t_2)) \not\vdash \exists p_1, p_2. p.\text{key} \mapsto k \ast p.\text{value} \mapsto v \ast
\]

\[
p.\text{left} \mapsto p_1 \ast p.\text{right} \mapsto p_2 \ast \text{Trep}(p_1, t_1) \ast \text{TBrep}(p_2, t_2)
\]

**Mutually Recursive Properties:**

\[
\Sigma; \Gamma; \text{TBrep}(p, r) \not\vdash \exists q. p \mapsto q \ast \text{Trep}(p, t)
\]

\[
\Sigma; \Gamma; \text{Trep}(p, N) \not\vdash p = null \land \text{emp}
\]

\[
\Sigma; \Gamma; \text{Trep}(p, T(t_1, k, v, t_2)) \not\vdash
\]

\[
p.\text{key} \mapsto k \ast p.\text{value} \mapsto v \ast \text{TBrep}(p, t_1) \ast \text{TBrep}(p, t_2)
\]

Figure 7.4: Representation Predicates for Binary Trees
Here, \( \text{Trep}(\mu, \nu) \) says a binary tree \( \nu \) is stored in memory and \( \mu \) is the root address. \( \text{TBrep}(\mu, \nu) \) says a binary tree \( \nu \) is stored in memory and \( \mu \) is a pointer which points to the root address (\( \mu \) itself is not the root address). The name “\( \text{Trep}(\_\_\_\_)_{} \)” stands for “tree representation predicate” and “\( \text{TBrep}(\_\_\_\_)_{} \)” stands for “tree-box representation predicate”. Figure 7.5 shows a comparison between them.

\[
\begin{align*}
T_5 &= \text{T}(N, 2, \text{\textquotedblright}b\text{\textquotedblright}, N) \\
T_7 &= \text{T}(N, 7, \text{\textquotedblright}g\text{\textquotedblright}, N) \\
T_{12} &= \text{T}(N, 12, \text{\textquotedblright}l\text{\textquotedblright}, N) \\
T_4 &= \text{T}(T_2, 4, \text{\textquotedblright}d\text{\textquotedblright}, N) \\
T_9 &= \text{T}(T_7, 9, \text{\textquotedblright}i\text{\textquotedblright}, T_{12}) \\
T_5 &= \text{T}(T_4, 5, \text{\textquotedblright}e\text{\textquotedblright}, T_9)
\end{align*}
\]

Figure 7.5: \( \text{TBrep}(\_\_\_\_)_{} \) and \( \text{Trep}(\_\_\_\_)_{} \).

In the proof rules in Figure 7.4, I use \( \mu.\text{key} \), \( \mu.\text{value} \), \( \mu.\text{left} \) and \( \mu.\text{right} \) to represent the address of field \( \text{key} \), \( \text{value} \), \( \text{left} \) and \( \text{right} \) when the head address of the whole C \text{struct} is \( \mu \). In other words, they represent \( \mu, \mu + 4, \mu + 8 \) and \( \mu + 12 \) respectively.

For verifying this specific BST insert program, only using \( \text{TBrep}(\_\_\_\_)_{} \) is enough. It is not necessary to have \( \text{Trep}(\_\_\_\_)_{} \) and to use related proof rules. However, I choose to use both representation predicates and their mutually recursive rules so that the program verification is more concise.

The verification target is described as the following Hoare triple:

\[
\{ \left[ [p] = p_0 \land [x] = x \land [v] = v \land \text{TBrep}(p_0, t_0) \right] \} \\
\text{insert}(p, x, v) \\
\{ \text{TBrep}(p_0, \text{ins}(t_0, x, v)) \}\]

(7.1)
Recall that it is the specification for *implementation correctness*. Specifically, it says: for any binary tree \( t_0 \), key \( x \) and value \( v \), if \( t_0 \) is stored in memory, then after executing this \texttt{insert} function the binary tree stored in memory is \( \text{ins}(t_0, x, v) \).

I explicitly use logic variables which are claimed to be equal to the denotation of program variables in the precondition. The purpose is not only to isolate program variables from separating conjuncts for the convenience of verification (for that purpose, isolating program variables in proofs is enough, it would be not necessary to isolate them in the specification), but also to mention these old values in the postcondition: the updated new tree should be stored at the original place and the insertion uses the original value of \( x \) and \( v \).

### 7.2 Magic wand for partial trees

The function body of \texttt{insert} is just one loop. We will need a loop invariant! As shown in Figure 7.2 the original binary tree can always be divided into two parts after every loop body iteration: one is a subtree \( t \) whose root is tracked by program variable \( p \) (that is, \([*p] \) is the address of \( t \)'s root node) and another part is a partial tree \( P \) whose root is identical with the original tree and whose hole is marked by address \([p] \).

The separation logic predicate for trees (also subtrees) is \texttt{TBrep}(.). We define the separation logic predicate for partial trees as follows. Given a partial tree \( P \), which is a function from binary trees to binary trees:

\[
\text{PTBrep}(r, i, P) \triangleq \forall t. \ (\text{TBrep}(i, t) \rightarrow \text{TBrep}(r, P(t)))
\]

This predicate has some important properties and we will use these properties in the verification of \texttt{insert}. Rules (7.2a) and (7.2b) show how single-layer partial trees are constructed. (7.2c) shows the construction of empty partial trees. (7.2d) shows
that a subtree can be filled in the hole of a partial tree. For example, in Figure 7.6, 
P_{5,9} is a partial tree with tree nodes 5, 4 and 2. The right subtree of node 5 is the hole in that partial. Filling that hole with a tree \( t_9 \) results in \( P_{5,9} t_9 = t_5 \). (7.2c) shows the composition of two partial trees. In Figure 7.6, merging \( P_{5,9} \) and \( P_{9,7} \) is \( P_{5,9} \circ P_{9,7} = P_{5,7} \); thus \( \Sigma; \Gamma; \text{PTBrep}(p_r, p_m, P_{5,9}) \ast \text{PTBrep}(p_m, p_i, P_{9,7}) \vdash \text{PTBrep}(p_r, p_i, P_{5,7}) \).

\[
\Sigma; \Gamma; \quad p \mapsto q \ast q.\text{key} \mapsto k \ast q.\text{value} \mapsto v \ast \text{TBrep}(q.\text{right}, t_2)) \\
\vdash \text{PTBrep}(p, q.\text{left}, \lambda t. T(t, k, v, t_2)))
\]

(7.2a)

\[
\Sigma; \Gamma; \quad p \mapsto q \ast q.\text{key} \mapsto k \ast q.\text{value} \mapsto v \ast \text{TBrep}(q.\text{left}, t_1)) \\
\vdash \text{PTBrep}(p, q.\text{right}, \lambda t. T(t_1, k, v, t)))
\]

(7.2b)

\[
\Sigma; \Gamma; \quad \text{emp} \vdash \text{PTBrep}(p, p, \lambda t. t)
\]

(7.2c)

\[
\Sigma; \Gamma; \quad \text{TBrep}(i, t) \ast \text{PTBrep}(r, i, P) \vdash \text{TBrep}(r, P(t))
\]

(7.2d)

\[
\Sigma; \Gamma; \quad \text{PTBrep}(p_1, p_2, P_1) \ast \text{PTBrep}(p_2, p_3, P_2) \vdash \text{PTBrep}(p_1, p_3, P_1 \circ P_2)
\]

(7.2e)

These properties are direct instances of the \textsc{wandQ-frame} proof rules. \textsc{wandQ-frame-intro} proves (7.2a), (7.2b) and (7.2c). \textsc{wandQ-frame-elim} proves (7.2d). \textsc{wandQ-frame-ver} and \textsc{wandQ-frame-refine} together prove (7.2e). The soundness of (7.2c), (7.2d) and (7.2e) do not even depend on the meaning of \text{TBrep}(\_,\_).

### 7.3 Implementation correctness proof

This section demonstrates a \textsc{wandQ-frame} verification of BST insert. Specifically, I verify the \text{insert} function with the loop invariant,

\[
\exists t, p, P. \ P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v) \land [p] = p \land [x] = x \land [v] = v \land \\
\text{TBrep}(p, t) \ast \text{PTBrep}(p_0, p, P)
\]

72
TBrep\((p_r, t_9) \ast PTBrep(p_r, p_i, P_{5,9})\) \\
\vdash TBrep(p_r, t_5)

PTBrep\((p_r, p_m, P_{5,9}) \ast PTBrep(p_m, p_i, P_{9,7})\) \\
\vdash PTBrep(p_r, p_i, P_{6,7})

Figure 7.6: Examples of rule (7.2d) and (7.2e).

It says, a partial tree \(P\) and a tree \(t\) are stored in disjoint pieces of memory, and if we apply the ins function to \(t\) locally and fill the hole in \(P\) with that result, then we will get the same as directly applying ins to the original binary tree \(t_0\).

The correctness of insert is based on the following two facts. First, the precondition of insert

\[[p] = p_0 \land [x] = x \land [v] = v \land TBrep(p_0, t_0)\]

implies this loop invariant because we can instantiate the existential variables \(t, p\) and \(P\) with \(t_0, p_0\) and \(\lambda t. t\) and apply property (7.2c). Second, the loop body preserves this loop invariant and every return command satisfies the postcondition of the whole
C function. Here is the loop body correctness proof (for conciseness, I omit \([x] = x \land [v] = v\) in all assertions).

1 \[\{\exists t, p, P. P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v) \land [p] = p \land \text{TBrep}(p, t) \land \text{PTBrep}(p_0, p, P)\}\]

2 Given \(t\) and \(p\). Assume \(P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v)\).

3 \[\{[p] = p \land \text{TBrep}(p, t) \land \text{PTBrep}(p_0, p, P)\}\]

4 \[\{\exists q. [p] = p \land p \mapsto q \land \text{Trep}(q, t) \land \text{PTBrep}(p_0, p, P)\}\]

5 \(\{}\)

6 Given \(q\).

7 \(\{[p] = p \land p \mapsto q \land \text{Trep}(q, t) \land \text{PTBrep}(p_0, p, P)\}\)

8 \(q = *p;\)

9 \(\{}\)

10 if (q == NULL) \(\{\)

11 \(\{[p] = p \land [q] = q \land q = \text{null} \land p \mapsto q \land \text{Trep}(q, t) \land \text{PTBrep}(p_0, p, P)\}\)

12 \(\{[p] = p \land [q] = \text{null} \land t = N \land p \mapsto \text{null} \land \text{Trep}(p_0, p, P)\}\)

13 Assume \(t = N\). Thus \(P(T(N, x, v, N)) = \text{ins}(t_0, x, v)\).

14 \(\{[p] = p \land p \mapsto \text{null} \land \text{Trep}(p_0, p, P)\}\)

15 \(q = (\text{struct tree} *) \text{surely\_malloc (sizeof *,q)};\)

16 \(q\rightarrow \text{key} = x; q\rightarrow \text{value} = v;\)

17 \(q\rightarrow \text{left} = \text{NULL}; q\rightarrow \text{right} = \text{NULL};\)

18 \(\text{p} = q;\)

19 \(\{[p] = p \land \text{TBrep}(p, T(N, x, v, N)) \land \text{PTBrep}(p_0, p, P)\}\)

20 \(\{[p] = p \land \text{TBrep}(p_0, P(T(N, x, v, N)))\}\)

21 \(\{\text{Trep}(p_0, \text{ins}(t_0, x, v))\}\)

22 \(\}\) return;

23 \}\) else \(\{\)

24 \(\{[p] = p \land [q] = q \land q \neq \text{null} \land p \mapsto q \land \text{Trep}(q, t) \land \text{PTBrep}(p_0, p, P)\}\)

25 \(\{\exists t_1, x_0, v_0, t_2. [p] = p \land [q] = q \land t = T(t_1, x_0, v_0, t_2) \land \}

26 \(\{p \mapsto q \land \text{Trep}(q, T(t_1, x_0, v_0, t_2)) \land \text{PTBrep}(p_0, p, P)\}\)
Given $t_1$, $x_0$, $v_0$ and $t_2$. Assume $t = T(t_1, x_0, v_0, t_2)$.

```plaintext
\{ [p] = p \land [q] = q \land p \mapsto q \ast q.\text{key} \mapsto x_0 \ast q.\text{value} \mapsto v_0 \ast \\
TBrep(q.\text{left}, t_1) \ast TBrep(q.\text{right}, t_2) \ast PTBrep(p_0, p, P) \}
```

```plaintext
int y = q\rightarrow key;
```

```plaintext
\{ [p] = p \land [q] = q \land [y] = x_0 \land p \mapsto q \ast q.\text{key} \mapsto x_0 \ast q.\text{value} \mapsto v_0 \ast \\
TBrep(q.\text{left}, t_1) \ast TBrep(q.\text{right}, t_2) \ast PTBrep(p_0, p, P) \}
```

```plaintext
\begin{align*}
\text{if} \ (x < y) & \\
\text{Assume} \ x < x_0. \text{ Thus } ins(t, x, v) = T(ins(t_1, x, v), x_0, v_0, t_2). \\
p = & q\rightarrow \text{left}; \\
\{ [p] = q.\text{left} \land [q] = q \land p \mapsto q \ast q.\text{key} \mapsto x_0 \ast q.\text{value} \mapsto v_0 \ast \\
TBrep(q.\text{left}, t_1) \ast TBrep(q.\text{right}, t_2) \ast PTBrep(p_0, p, P) \}
\end{align*}
```

```plaintext
\begin{align*}
\text{else if} \ (y < x) & \\
\text{Assume} \ x_0 < x. \\
p = & q\rightarrow \text{right}; \\
\{ P(T(t_1, x_0, v_0, \text{ins}(t_2, x, v))) = \text{ins}(t_0, x, v) \land [p] = q.\text{right} \\
\ast TBrep(q.\text{right}, t_2) \ast PTBrep(p_0, q.\text{right}, \lambda \hat{t}. P(T(\hat{t}, x_0, v_0, \hat{t}))) \}
\end{align*}
```

```plaintext
\begin{align*}
\text{else} & \\
\text{Assume} \ x_0 = x. \text{ Thus } ins(t, x, v) = T(t_1, x, v, t_2). \\
p \mapsto \text{value} = v; \\
\{ TBrep(p, T(t_1, x, v, t_2)) \ast PTBrep(p_0, p, P) \}
\end{align*}
```

```plaintext
\begin{align*}
\text{else} \{ \\
\text{Assume} \ x_0 = x. \text{ Thus } ins(t, x, v) = T(t_1, x, v, t_2). \\
p \mapsto \text{value} = v; \\
\{ TBrep(p_0, P(T(t_1, x, v, t_2))) \}
\end{align*}
```

```plaintext
\begin{align*}
\text{return}; \\
\}
\}
```

This loop body has four branches: two of them end with return commands and the other two end normally.

In the first branch (line 21), the inserted key does not appear in the original tree. This branch ends with a return command. We show that the program state at that point satisfies the postcondition of the whole function body (line 20). The transition from line 18 to line 19 uses rule (7.2d)—filling the hole of a partial tree with a complete tree in separation logic.

The second branch contains only one command at line 31. We re-establish our loop invariant in the end this branch (line 34). The transition from line 32 to line 33 is due to rule (7.2a)—single layer partial tree construction. And the transition from line 33 to line 34 is due to rule (7.2e)—merging two partial trees together.

The third branch at line 37 is like the second, and the last branch is like the first one.

In summary, the partial tree $P$ is established as an empty partial tree ($\lambda \mathring{t} \mathring{t}$) in the beginning. The program merges one small piece of subtree $t$ into the partial tree in each iteration of the loop body. Finally, when the program returns, it establishes a local insertion result (ins($t, x, v)$) and fills it in the hole of that partial tree—we know the result must be equivalent with directly applying insertion to the original binary tree. The diagrams above illustrate the situations of these four branches and our proof verifies this process.
7.4 Other BST operations

I also verify C implementations of BST delete and look-up operation with the magic-wand-as-frame technique.

```c
void *lookup (tree *p, int x) {
    void *v;
    while (p != NULL) {
        int y = p->key;
        if (x < y)
            p = p->left;
        else if (y < x)
            p = p->right;
        else {
            v = p->value;
            return v;
        }
    }
    return NULL;
}

void turn_left(tree *l, struct tree *l, struct tree *r) {
    struct tree *mid;
    mid = r->left;
    l->right = mid;
    r->left = l;
    *l = r;
}
```
void pushdown_left (treebox t) {
    struct tree *p, *q;
    for(;;) {
        p = *t; q = p->right;
        if (q==NULL) {
            q = p->left; *t = q;
            freeN(p, sizeof (*p)); return;
        } else {
            turn_left(t, p, q); t = &q->left;
        }
    }
}

void delete (treebox t, int x) {
    struct tree *p;
    for(;;) {
        p = *t;
        if (p==NULL) {
            return;
        } else {
            int y = p->key;
            if (x<y)
                t = &p->left;
            else if (y<x)
                t = &p->right;
            else {
                pushdown_left(t);
                return;
            }
        }
    }
}
In the verification of BST delete, I also use PTBrep(., . , . ) to describe partial trees and use rules (7.2a, 7.2e) to complete the proof. In the verification of BST look-up, because the C program does not use pointer-to-pointer-to-tree-nodes, I only use Trep(., . ) to describe trees and define the following predicate (the partial tree representation predicate) to describe partial trees:

\[ \text{PTrep}(r, i, P) \triangleq \forall t. \ (\text{Trep}(i, t) \rightarrow \text{Trep}(r, P(t))) \]

In comparison, it is very similar with the definition of PTBrep(r, i, P):

\[ \text{PTBrep}(r, i, P) \triangleq \forall t. \ (\text{TBrep}(i, t) \rightarrow \text{TBrep}(r, P(t))) \]

For verifying BST look-up, I prove the following counterparts of (7.2a-7.2e):

\[ \Sigma; \Gamma; \ p.\text{key} \mapsto k \ * \ p.\text{value} \mapsto v \ * \ p.\text{left} \mapsto l \ * \ p.\text{right} \mapsto r \ *
\text{Trep}(r, t_2)) \vdash \text{PTrep}(p, l, \lambda t. \ T(t, k, v, t_2)) \]
\[ (7.3a) \]

\[ \Sigma; \Gamma; \ p.\text{key} \mapsto k \ * \ p.\text{value} \mapsto v \ * \ p.\text{left} \mapsto l \ *
\text{Trep}(l, t_1)) \vdash \text{PTrep}(p, r, \lambda t. \ T(t_1, k, v, t)) \]
\[ (7.3b) \]

\[ \Sigma; \Gamma; \ emp \vdash \text{PTrep}(p, p, \lambda t. \ t) \]
\[ (7.3c) \]

\[ \Sigma; \Gamma; \ \text{Trep}(i, t) \ * \ \text{PTrep}(r, i, P) \vdash \text{Trep}(r, P(t)) \]
\[ (7.3d) \]

\[ \Sigma; \Gamma; \ \text{PTrep}(p_1, p_2, P_1) \ * \ \text{PTrep}(p_2, p_3, P_2) \vdash \text{PTrep}(p_1, p_3, P_1 \circ P_2) \]
\[ (7.3e) \]

I claimed in section 7.2 that the soundness of (7.2c, 7.2d) and (7.2e) does not depend on the meaning of TBrep(., . ). Here is evidence: (7.3c, 7.3d) and (7.3e) have exactly the same forms as (7.2c, 7.2d) and (7.2e). In section 10.3, I will reformulate this fact (in a more precise way) by a *parameterized* soundness theorem.
Chapter 8

Improving ramification

Hobor and Villard [30] demonstrate the ramification rule:\[1\]

\[\text{RAMIF:}\]

If \(\Sigma; \Gamma \vdash \{P_L\} \; c \; \{Q_L\}\), \(\Sigma; \Gamma; P_G \vdash P_L \ast (Q_L \rightarrow Q_G)\) and any nonaddressable variable which is modified in \(c\) does not freely occur in \((Q_L \rightarrow Q_G)\)

then \(\Sigma; \Gamma \vdash \{P_G\} \; c \; \{Q_G\}\)

They apply this proof rule in graph algorithm verification. Unlike FRAME, RAMIF does not require the pre/postcondition in the global triple to be in the special forms \(P \ast F\) and \(Q \ast F\). Instead, a side condition for local and global pre/postconditions must proved: \(\Sigma; \Gamma; P_G \vdash P_L \ast (Q_L \rightarrow Q_G)\). I call this side condition the \textit{ramification premise}.

\[1\text{In their Hoare logic, there is no return condition, break condition or continue condition.}\]
Hobor and Villard show that ramification premises can be proved compositionally,

**RAM-HOR:**

If $\Sigma; \Gamma; P_G \vdash P_L * (Q_L \rightarrow Q_G)$ and

$\Sigma; \Gamma; P'_G \vdash P'_L * (Q'_L \rightarrow Q'_G)$

then $\Sigma; \Gamma; P_G * P'_G \vdash P_L * P'_L * (Q_L * Q'_L \rightarrow Q_G * Q'_G)$.

Actually, ramification can be treated as a special application of WAND-FRAME. The **RAMIF** rule can be proved by FRAME, HOARE-CONS and WAND-FRAME-ELIM:

\[
\begin{array}{c}
\{P_G\} \\
\{P_L * (Q_L \rightarrow Q_G)\} \\
\{P_L\} \\
\{Q_L\} \\
\{Q_L * (Q_L \rightarrow Q_G)\} \\
\{Q_G\}
\end{array}
\]

That is, introducing a wand-frame once and eliminating it once. Also, **RAM-HOR** is a direct corollary of WAND-FRAME-HOR:

Assume $\Sigma; \Gamma; P_G \vdash P_L * (Q_L \rightarrow Q_G)$ and

$\Sigma; \Gamma; P'_G \vdash P'_L * (Q'_L \rightarrow Q'_G)$

then $\Sigma; \Gamma; P_G * P'_G \vdash (P_L * (Q_L \rightarrow Q_G)) * (P'_L * (Q'_L \rightarrow Q'_G))$

\[
\vdash P_L * P'_L * (Q_L \rightarrow Q_G) * (Q'_L \rightarrow Q'_G)
\]

\[
\vdash P_L * P'_L * (Q_L * Q'_L \rightarrow Q_G * Q'_G)
\]
Magic-wand-as-frame is more flexible than ramification. A magic-wand-as-frame proof may use a wand expression between separating conjuncts of local and global postconditions. But ramification only uses magic wand to connect the whole local and global postconditions.

The original ramification had two major limitations: it does not interact well with existential quantifiers or with modified nonaddressable program variables. \textsc{wandQ-frame} rules and \textsc{wandP-frame} rules respond to these two problems.

Specifically, when local and global postconditions are existentially quantified, we can apply the following variant of \textsc{ramif}. It is a corollary of \textsc{wandQ-frame} rules and its ramification premise is still compositional.

\textsc{ramifQ}:

If \( \Sigma; \Gamma \vdash \{ P_L \} \ c \ \{ \exists x.Q_L(x) \} \), \( \Sigma; \Gamma; P_G \vdash P_L * \forall x.(Q_L(x) \rightarrow Q_G(x)) \) and any nonaddressable variable which is modified in \( c \) does not freely occurs in \( (Q_L(x) \rightarrow Q_G(x)) \)

then \( \Sigma; \Gamma \vdash \{ P_G \} \ c \ \{ \exists x.Q_G(x) \} \)

\textsc{ramQ-hor}:

If \( \Sigma; \Gamma; P_G \vdash P_L * \forall x.(Q_L(x) \rightarrow Q_G(x)) \) and

\( \Sigma; \Gamma; P'_G \vdash P'_L * \forall x.(Q'_L(x) \rightarrow Q'_G(x)) \)

then \( \Sigma; \Gamma; P_G * P'_G \vdash P_L * P'_L * \forall x.(Q_L(x) * Q'_L(x) \rightarrow Q_G(x) * Q'_G(x)) \).

82
When modified nonaddressable program variables freely occur in local and global postconditions, we can apply the ramification counterpart of WANDP-FRAME rules.

**RAMIFP:**

If $\Sigma; \Gamma \vdash \{P_L\} c \{Q_L\}$, $\Sigma; \Gamma; P_G \vdash P_L \ast [\vec{x}](Q_L \rightarrow Q_G)$ and $\vec{x}$ are all modified variables in $c$

then $\Sigma; \Gamma \vdash \{P_G\} c \{Q_G\}$
Part II

VST-Floyd: proved sound separation-logic-based verification tool
Chapter 9

Design Principles

In Part I, I focused on using separation logic to verify program correctness. From this chapter on, I begin to introduce VST-Floyd, which helps us to formalize these separation logic proofs in Coq.

Specifically, VST-Floyd provides a lemma and tactic library for proving Hoare triples valid with respect to CompCert Clight’s operational semantics. Elementary proof rules like `hoare-seq`, `hoare-load` and etc. were proved sound in previous work using Verifiable C’s step indexed semantic model. VST-Floyd builds a user-friendly interface based on Verifiable C.

VST-Floyd has two main design principles. One is to maximize users’ convenience in applying the program logic (the separation Hoare logic of Verifiable C). For example in section 7.2, the correctness of BST-insert’s loop body is ensured by the following Hoare triple:
\[ x; v; t_0; p_0; t; p; q; P; P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v); \quad \Delta \vdash \]
\[
\{ [x] = x \land [v] = v \land [p] = p \land p \mapsto q \} \ast \text{Trep}(q, t) \ast \text{PTBrep}(p_0, p, P) \}
\]
\[ q = *p; \]
\[
\begin{align*}
\text{if} (q==\text{NULL}) \{ \\
\quad q = (\text{struct tree } *) \text{ surely malloc}(\text{sizeof } *p); \\
\quad q\rightarrow\text{key}=x; q\rightarrow\text{value}=v; \\
\quad q\rightarrow\text{left}=\text{NULL}; q\rightarrow\text{right}=\text{NULL}; \\
\quad *p=q; \\
\quad \text{return}; \\
\} \text{ else } \{ \\
\quad \text{int } y = q\rightarrow\text{key}; \\
\quad \text{if } (x<y) \\
\quad \quad p= &q\rightarrow\text{left}; \\
\quad \text{else if } (y<x) \\
\quad \quad p= &q\rightarrow\text{right}; \\
\quad \text{else} \{ \\
\quad \quad q\rightarrow\text{value}=v; \\
\quad \quad \text{return}; \\
\quad \} \\
\} \Bigg\}
\]
\[
\exists t, p, P. P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v) \land \\
\{ [x] = x \land [v] = v \land [p] = p \land \text{Trep}(p, t) \ast \text{PTBrep}(p_0, p, P), \\
[\bot, \bot, \text{Trep}(p_0, \text{ins}(t_0, x, v))] \}
\]

Straightforwardly, we should prove it by the sequence rule (HOARE-SEQ), a Hoare triple for the first assignment command and a Hoare triple for the big if command. We know that for any context Σ and assumption set Γ (this can be formally proved by SEMAX-LOAD and HOARE-CON):
Σ; Γ; Δ ⊢

\{ [x] = x \land [v] = v \land [p] = p \land p \mapsto q \ast \text{Trep}(q, t) \ast \text{PTBrep}(p_0, p, P) \}

q = *p;

\{ [x] = x \land [v] = v \land [p] = p \land [q] = q \land p \mapsto q \ast \text{Trep}(q, t) \ast \text{PTBrep}(p_0, p, P) \}

Thus, it suffices to prove:

x; v; t_0; p_0; t; p; q; P; P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v); Δ ⊢

\{ [x] = x \land [v] = v \land [p] = p \land [q] = q \land p \mapsto q \ast \text{Trep}(q, t) \ast \text{PTBrep}(p_0, p, P) \}

The if command

\begin{align*}
\exists t, p, P. & P(\text{ins}(t, x, v)) = \text{ins}(t_0, x, v) \land \\
[x] = x & \land [v] = v \land [p] = p \land [q] = q \land p \mapsto q \ast \text{Trep}(q, t) \ast \text{PTBrep}(p_0, p, P), \\
[\bot, \bot, \text{TBrep}(p_0, \text{ins}(t_0, x, v))] &
\end{align*}

VST-Floyd provides a tactic \texttt{forward} which will reduce the original Hoare triple proof goal to this one above in Coq. In other words, \texttt{forward} will take away the first assignment command in the Hoare triple in this case, and replace the origin precondition with that assignment command’s strongest postcondition.

\textbf{The other design principle of VST-Floyd is: users should be able to flexibly use Coq’s built-in tactic systems to derive domain-specific conclusions.}

Coq’s developers have been enhancing its tactic system for 30 years. Coq’s tactics are now very convenient to use in building proofs. For example, Coq’s users can apply proved domain-specific theorems on assumptions. Coq’s users can rewrite a term in Coq’s conclusion if an equation has been proved. Even some useful domain-specific decision procedures have been built into Coq’s standard library. For example, \texttt{omega} solves integer linear programming. We do not want to rebuild all of them in VST-Floyd. Instead, we let users use Coq’s built-in tactics as much as possible. To support this goal, our canonical form (see Chapter 11) and our tactics (see Chapter 13) are organized to minimize the occurrence of program variables in pre/postconditions—most
expressions in assertions appear as ordinary Coq values which can be manipulated by Coq’s built-in tactics.

VST-Floyd is built up by not only my personal effort. Andrew W. Appel, Lennart Beringer, Samuel Gruetter and other members of VST research group contributed a lot to this verification tool.
Chapter 10

Separation logic predicates in VST-Floyd

In this chapter, I introduce how separation logic predicates and assertions are formalized in VST-Floyd.

10.1 Background: shallowly embedded separation logic

Verifiable C's assertion language is shallowly embedded into Coq. In other words, every assertion $P$ is a predicate on stack-heap pairs. Specifically, a stack is a function from nonaddressable variables to values and a heap is a partial function from addresses to values.

\[1\] The readers can understand the type of $P$ as stack $\rightarrow$ heap $\rightarrow$ Prop. But actually, this predicate must be monotonic w.r.t. the step indexing, i.e. we define it Coq as a dependent pair of a predicate and a proof of monotonicity [5 Part V].
All logical connectives are formalized semantically instead of syntactically. For example,

\[ P \land Q \triangleq \lambda s, h. P(s, h) \land Q(s, h) \]
\[ P * Q \triangleq \lambda s, h. \exists h_1, h_2. \oplus(h_1, h_2, h) \land P(s, h_1) \land Q(s, h_2) \]

Here, the connectives on the left side are logical connectives of our assertion language while the connectives on the right side belong to metalanguage and describe the underlying memory model directly.

10.1.1 Concrete-typed quantifiers

Quantifiers in shallow embedding can be very easily defined. Recall that all quantifiers in our assertion language are typed; we can quantify over natural numbers, list of scalar values, functions from a fixed type to another, etc. In the assertion

\[ \forall x : A. P(x) \]

a Coq type \(A\) is directly used as a type in our assertion language; \(P\) is a Coq function from \(A\) to assertion; and the universal quantifier in the assertion language can be directly defined as follows:

\[ \forall x : A. P(x) ::= \lambda s : \text{stack}. \lambda h : \text{heap}. \text{for any } x : A, \ P(x, s, h) \]

Again, the connectives on the right side belong to metalanguage.

10.1.2 Entailments and Hoare triples

As described in chapter 2 an entailment between two separation logic assertions is formalized as a Coq proposition. This formalization is also shallowly embedded: the
Coq proposition “derives \( P \rightarrow Q \)” or in notation “\( P \vdash Q \)” is defined as (again, this definition still belongs to metalanguage):

\[
\forall s, h. P(s, h) \rightarrow Q(s, h).
\]

Also, the Hoare triple \( \{ P \} c \{ Q \} \) is semantically formalized as a Coq proposition:

“semax \( \Delta \) \( P \) \( c \) (\( Q_1, [Q_2, Q_3, Q_4] \))” can be (informally) interpreted as, for any program state \((s_1, h_1)\), if it satisfies \( P \), then (1) it is safe to execute \( c \) from it; (2) if such execution terminates with a program state \((s_2, h_2)\) normally, by \texttt{break}, by \texttt{continue} or by \texttt{return}, then \((s_2, h_2)\) should satisfy \( Q_1, Q_2, Q_3 \) or \( Q_4 \) respectively.

\subsection*{10.1.3 Unlifted logic: stack-independent heap predicates}

In the separation logic of Verifiable C, assertions independent of nonaddressable variable values (the stack) are treated differently. For example, if \( p \) and \( v \) are values, then \( p \mapsto v \) is a predicate independent of stacks. In comparison, \([x] \mapsto 0\) is not independent of stacks, since it relies on the current value of the local variable \( x \).

The fact that an assertion \( P \) is independent of stacks could be described by the following metalogic proposition:

\[
\forall s_1, s_2, h. P(s_1, h) \iff P(s_2, h)
\]

Verifiable C chooses a more straightforward way: those assertions independent of stacks are just predicates over heaps! They also form a separation logic, i.e.

\[
P \star Q \triangleq \lambda h. \exists h_1, h_2. h_1 \oplus h_2 = h \land P(h_1) \land Q(h_2)
\]
We call this separation logic unlifted and we call the previous one lifted. The lifted separation logic and unlifted separation logic have the following connection.

\[ P \land Q = \lambda s : \text{stack}. P(s) \land Q(s) \]
\[ P * Q = \lambda s : \text{stack}. P(s) * Q(s) \]
\[ \exists x : A. P(x) = \lambda s : \text{stack}. \exists x : A. P(x, s) \]
\[ \forall x : A. P(x) = \lambda s : \text{stack}. \forall x : A. P(x, s) \]

Here, all connectives on the left side are logical connectives of the lifted logic while function abstraction symbol “\( \lambda s : \text{stack} \)” is a metalogic symbol and all other connectives on the right side belong to the unlifted logic. The situation with other connectives is similar; I omit them here.

For its spatial predicates, Verifiable C always defines unlifted predicates like \( p \mapsto v \) (which is a predicate over heaps) first, then defines lifted predicates or assertions based on that. For example, \( [x] \mapsto [y] \) is a lifted assertion, i.e. a predicate over stacks and heaps. It is defined by its unlifted version as follows:

\[ \lambda s, h. ([x](s) \mapsto [y](s))(h) \]

Here, given a specific stack \( s \), \([x](s)\) and \([y](s)\) are pure values, and \([x](s) \mapsto [y](s)\) is an unlifted assertions, i.e. a predicate over heaps.

This unlifted logic is not only useful in Verifiable C’s Hoare logic soundness proofs. In program verification, it is particularly useful when the assertions are written in a program-variable-isolated form. I will come back to that in Chapter \( \square \)
10.2 Separation logic predicates for aggregate types

Verifiable C formalizes two spatial predicates in its separation logic: mapsto and memory.block. Specifically,

\[ \text{mapsto } \pi t p v \]

describes a singleton heap with value \( v \) stored at address \( p \). The parameter \( \pi \) represents the read/write permission owned and \( t \) indicates the C type of that address. Since this predicate only describes singleton heaps, \( t \) must be scalar C types or pointers. Besides,

\[ \text{memory.block } \pi n p \]

describes a size-\( n \) heap with a contiguous segment of addresses starting from \( p \). Again the parameter \( \pi \) represents the read/write permission of this heap. The permission \( \pi \) does not play an important role in this thesis. I will usually omit it.

In Reynolds’s original version of separation logic [18], he used abbreviations like \( p \mapsto a, b \) for conciseness, to mean \( p \mapsto a \ast p + 1 \mapsto b \). VST-Floyd formalizes such separation logic predicates for structured data described by C aggregate types in order to improve the usability of Verifiable C beyond atomic mapsto predicates \( p \mapsto v \).

Specifically, “data.at \( \pi t v p \)” says that a datum \( v \) of C type \( t \) is stored at address \( p \). Here, \( t \) is the AST (syntactic description) of a C-language type expression, including integers, floats, pointers, struct, union and array. I will usually omit the permission argument \( \pi \) in this thesis and write \( p \mapsto t v \) in short.

The assertion \( p \mapsto t v \) is dependently typed: the type of \( v \) depends on the value of \( t \). When \( t \) is a struct, \( v \) is a tuple. For example\(^2\).

\(^2\)Here, I use “\( \approx \)” instead of equality or logical equivalence because eventually, they are not logically equivalent. The left side will be defined to be stronger than the right side (see section 10.2.2).
\textbf{struct} IntPair \{ \textbf{int} \ fst; \textbf{int} \ snd; \};

\begin{align*}
p & \mapsto_{\text{IntPair}} a, b \\ & \approx \ p \mapsto a \ast p + 4 \mapsto b
\end{align*}

An integer occupies 4 bytes, so the second field is stored at $p + 4$.

Similarly, when $t$ is a \textbf{union}, $v$’s type is a sum type. When $t$ is an array, $v$ is a list. For example,

\begin{align*}
p & \mapsto_{\text{IntPair}[3]} [a; b; c] \\ & \approx \ p \mapsto a \ast p + 4 \mapsto b \ast p + 8 \mapsto c
\end{align*}

\textbf{Struct}, \textbf{union} and array can be nested. For example, the following predicate describes an array of \textbf{struct} type.

\begin{align*}
p & \mapsto_{\text{IntPair}[3]} [(a_1, a_2); (b_1, b_2); (c_1, c_2)]
\end{align*}

In principle, there is no need for the abbreviation $p \mapsto a, b$; one could write a separation conjunction of primary mapsto predicates. But without this abbreviation, when verifying an $n$-statement basic block that manipulates an $n$-field structure, each assertion (precondition of each statement) will have $n$ spatial conjuncts on which to do operations that are linear-time (or often quadratic) in $n$, leading to quadratic (or cubic) time for the whole block. Many-field structures are sufficiently common in real programs that this inefficiency is a significant problem—in addition to the notational inconvenience.

In this section, I will first introduce the properties that \texttt{data.at} should satisfy (subsection 10.2.1). Then I will find out a formalization that follows these properties (subsection 10.2.2) and demonstrate its Coq implementation in the end (subsection 10.2.3).
10.2.1 Proof theory

Data.at, field.at and field.address form the syntax for C aggregate types. The spatial predicate

\[ \text{field.at} \pi t \xrightarrow{f} p v \quad \text{(in short} \quad p \xrightarrow{t} v) \]
says, starting from address \( p \), which is the base address in memory to store a datum of type \( t \), following the path \( f \) of field-selection/array-indexing will arrive at a memory datum \( v \) of type “\( t.f \)”. The \textit{nested field} \( f \) is a path of general fields (a struct field, a union field or an array subscript). Additionally,

\[ \text{field.address} \ t \xrightarrow{f} p \]
represents the starting address of nested field \( f \) when the base address of the entire aggregate type \( t \) is at \( p \). For conciseness, I will use \( p.f \) to represent “\text{field.address} \ t \xrightarrow{f} p” when \( t \) can be inferred from the context.

The separation logic proof rules for C aggregate types contain a series of unfolding rules, a reroot theorem and an proof rule about memory.block, shown in Figure 10.1. The unfolding rules transform data.at into separating conjunctions of ordinary maps-to predicates. Entailment (10.1) says that data.at is field.at with empty field path. Entailment (10.2) says that field.at on elementary types (integers, floating point numbers, pointers) is equivalent to an ordinary maps-to predicate with an offset. Entailments (10.3), (10.4), and (10.5) are single-layer unfolding rules. Here, I overload the dot notation and use \( t.f \) to represent the type of field \( f \) in \( t \). I use the “Space” predicate (a special memory.block) to implement C’s alignment rules for struct fields and union fields.

Entailment (10.6) is the reroot theorem: its left side is a predicate on an internal node of a “tree” and the right side treats the internal node as a root. For example,

\[ \Sigma; \Gamma; \quad p \xrightarrow{[\text{fst}]} \text{IntPair} \ a \quad \not\vdash \quad p.\text{fst} \xrightarrow{\text{int}} a. \]
Finally, entailment (10.7) demonstrates the transition between `memory_block` and `data_at`.

### 10.2.2 Design choices

The proof theory in Figure 10.1 is very elegant—every rule is an unconditional logical equivalence, thus it is very convenient to use as arguments of Coq’s `rewrite` tactic. However, constructing a formal interpretation of `data_at` and `field_at` to validate these proof rules is not straightforward.

One problem is that we have to compromise with CompCert’s formalization of addresses. A 32-bit CompCert address $p$ is a pair of memory block identifier $b$ and 32-bit offset $n$. Pointer arithmetic is also 32-bit, i.e.

$$\text{Pointer}(b, n) + m = \text{Pointer}(b, (n + m) \mod 2^{32}).$$
That means, it is possible that \( p \) points to the last byte in a memory block but \((p + 1)\) points to the first byte in the same block (e.g. when \( p = \text{Pointer}(b, 2^{32} - 1) \)). Thus, making rule (10.3) sound is nontrivial.

\[
\Sigma; \Gamma; \quad p \xrightarrow{f} v \quad \vdash \quad \bigstar_{f \in \mathcal{T}} \left( p \xrightarrow{f} v.f \ast \text{Space}(f, p) \right)
\]

if \( t.\vec{f} \) is a nonempty struct \( (10.3) \)

My solution is: \text{field.at} should ensure its arguments satisfy a compatibility condition, which is called

\[
\text{field-compatible} \quad t \xrightarrow{f} p
\]

\text{Field-compatible} is a pure predicate saying that

1. \( p \) is a legal starting address for type \( t \), i.e. (1a) there is enough space in the memory block from \( p \) to store a datum of type \( t \) and (1b) \( p \) is a multiple of the alignment of type \( t \);

2. \( \vec{f} \) is a legal nested field of \( t \), i.e. (2a) \text{struct} or \text{union} fields in the path are fields in structure definition and (2b) array subscripts are in range.

It is worth noticing that \text{field-compatible} is a global compatibility condition. It not only requires compatibility between \( p \) and that single field described by \( \vec{f} \) but also requires compatibility between \( p \) and the whole type \( t \).

This design validates the unfolding rules but causes further problems in the soundness of the reroot theorem (entailment (10.6)). Its left side is a \text{field.at} which requires compatibility with the whole type \( t \). In contrast, the right side of it is a \text{data.at}, which only requires compatibility of the field type. To address this problem, we refine the
definition of “field.address $t \rightarrow_f p$” (or in short $p.\rightarrow_f$) as follows:

$$p.\rightarrow_f = p + \delta(\rightarrow_f) \quad \text{if field.compatible}(t, \rightarrow_f, p)$$

$$p.\rightarrow_f = \text{Vundef} \quad \text{otherwise}$$

Here, \text{Vundef} is a Coq term defined by Clight for representing illegal or uninitialized values (see footnote [1] in section 2.2); $\delta(\rightarrow_f)$ represents the offset of $\rightarrow_f$.

### 10.2.3 Coq implementation

In Coq, the predicate $p \mapsto v$ (or \text{data.at}) is typed as follows.

\[
\text{reptype} : \text{Clight.type} \rightarrow \text{Type} \n\]

\[
\text{data.at} : \forall t : \text{Clight.type}, \text{reptype} t \rightarrow \text{val} \rightarrow \text{pred heap} \n\]

The function \text{reptype} means “representation type”; it translates from a syntactic description of a C type to a Coq Type. A C array is represented as a Coq list, a \text{struct} is represented as a tuple, and a \text{union} is represented as a sum. Figure 10.2 shows some examples. By (\text{pred heap}) in Coq, I mean predicates over heaps—I follow the practice of Verifiable C and define \text{data.at} first as an unlifted predicate.

\[
\text{reptype}(\text{int}) = \text{val} \\
\text{reptype}(\ast \text{int}) = \text{val} \\
\text{reptype}(\text{struct IntPair}) = \text{val} \times \text{val} \\
\text{reptype}(\text{struct IntPair}[3]) = \text{list} (\text{val} \times \text{val})
\]

Figure 10.2: Examples of representation types

\footnote{Andrew W. Appel designed this feature in 2015.}
Similarly, \( p \vdash_{t} v \) (or field.at) is typed as follows.

**Inductive** \( \text{gfield} : \text{Type} := \)

| ArraySubsc : \( \forall i : \mathbb{Z}, \text{gfield} \) |
| StructField : \( \forall i : \text{ident}, \text{gfield} \) |
| UnionField : \( \forall i : \text{ident}, \text{gfield} \).

\( \text{nested\_field\_type} : \text{Clight\_type} \rightarrow \text{list gfield} \rightarrow \text{Clight\_type} \)

\( \text{field\_at} : \forall (t : \text{Clight\_type}) (\text{path} : \text{list gfield}), \)

\( \text{retype} (\text{nested\_field\_type} t \text{ path}) \rightarrow \text{val} \rightarrow \text{pred heap} \)

Here, \( \text{gfield} \) means “general field” and a general field can be a struct field, a union field or an array subscript. Thus a list of \( \text{gfield} \)s represents a path from a root type to a field type. The function “\( \text{nested\_field\_type} t \rightarrow f \)” is the Coq formalization of \( t.\rightarrow f \), i.e. it computes that field type from the given root type \( t \) and the given path of general fields \( \rightarrow f \).

I first define an auxiliary predicate \( p \vdash_{t} v \), or \( \text{data\_at\_rec} t v p \), as a recursive function in Coq. Then we define \( \text{field\_at} \) as an instance of \( \text{data\_at\_rec} \) and define \( \text{data\_at} \) as \( \text{field\_at} \) with empty field path. Specifically:

\[
\begin{align*}
\quad p \vdash_{t} v & \quad ::= \quad p \rightarrow v & \quad \text{if } t \text{ is integer, float, a pointer} \\
\quad p \vdash_{\text{struct } t} v & \quad ::= \quad \star_{f \in t} \left( p + \delta(f) \vdash_{t, f} v.f * \text{Space([f], p)} \right) \\
\quad p \vdash_{\text{union } t \{ f : v \}} v & \quad ::= \quad p \vdash_{t, f} v * \text{Space([f], p)} \\
\quad p \vdash_{t[n]} v & \quad ::= \quad \star_{0 \leq i < n} \left( p + i \cdot \text{sizeof}(t) \vdash_{t} v_{i} \right) \\
\quad p \vdash_{t} v & \quad ::= \quad p \vdash_{t} v & \quad \land \text{field\_compatible}(t, \rightarrow f, p) \\
\quad p \vdash_{t} v & \quad ::= \quad p \left[ \rightarrow_{t} v \right]
\end{align*}
\]

Here, when \( v \) is a tuple, \( v.f \) represents the element indexed by \( f \). When \( v \) is a list, \( v_{i} \) represents the \( i \)-th element of \( v \).
We could let \texttt{data.at} be defined as a conjunction of \texttt{data.at.rec} and \texttt{field.compatible} directly and let the reroot equation be the definition of \texttt{field.at}. However, treating \texttt{data.at} as \texttt{field.at} with empty path enables us to handle both constructions uniformly in our tactics.

\texttt{Reptype} and \texttt{data.at.rec} are implemented as Coq functions by recursion on Clight types\footnote{In CompCert 2.4 and earlier versions, the Clight type definition is a Coq inductive type. However, from CompCert 2.5, \texttt{struct} and \texttt{union} types are represented by name instead of by structure. Specifically, every Clight program is associated with a \texttt{composite.env}. A \texttt{composite.env} is a dictionary mapping every \texttt{struct}/\texttt{union} name to a list of all its fields. The meaning of a \texttt{struct} or a \texttt{union} needs to be interpreted by looking it up in the dictionary. From then on, \texttt{reptype} and \texttt{data.at.rec} are no longer Coq functions recursive on Coq inductive structure. The CompCert developers accepted our suggestion that every type should be tagged with a rank, which is a natural number. The ranking system ensures that the rank of a \texttt{struct} type is the max rank of its fields plus one; the rank of a \texttt{union} type is the max rank of its fields plus one; the rank of an \texttt{array} type is the rank of its element type plus one. The rank of elementary types (including pointers) is zero. Our current definition of \texttt{reptype} and \texttt{data.at.rec} are recursive functions on this rank.} \texttt{Nested.field.type} and \texttt{nested.field.offset} ($\delta(f)$) are implemented as Coq functions by recursion on the path.
10.3 Separation logic predicates for inductive data structures

Besides predicates for C aggregate types, it is also convenient to formalize domain-specific separation logic predicates. This section uses binary trees as an example, showing their formalization and proving their important properties.

10.3.1 Example: binary trees

Binary trees with keys and values, as a set of math objects, are already formalized in *Verified Functional Algorithms* [1] as an inductive data type in Coq.

Section TREES.

Variable V : Type. Variable default: V.

Definition key := nat.

Inductive tree : Type :=
| E : tree |
| T: tree -> key -> V -> tree -> tree.

End TREES.

We can define separation logic predicates as recursive Coq functions over this inductive structure. Here is the formalization of the separation logic predicates Trep( _, _) (called tree_rep in Coq) and TBrep( _, _) (called treebox_rep in Coq) that I use in Chapter 7.
**Fixpoint** tree_rep (t: tree val) (p: val) : pred heap :=

match t with
| N ⇒ !!(p=nullval) && emp
| T a x v b ⇒
    EX pa:val, EX pb:val,
    data_at Tsh t.struct_tree (Vint (Int.repr (Z.of_nat x)),(v,(pa,pb))) p *
    tree_rep a pa * tree_rep b pb
end.

**Definition** treebox_rep (t: tree val) (b: val) :=

EX p: val, data_at Tsh (tptr t.struct_tree) p b * tree_rep t p.

**Lemma** treebox_rep_spec: forall (t: tree val) (b: val),

treebox_rep t b =

EX p: val,

data_at Tsh (tptr t.struct_tree) p b *

match t with
| N ⇒ !!(p=nullval) && emp
| T l x v r ⇒
    field_at Tsh t.struct_tree [StructField .key] (Vint (Int.repr (Z.of_nat x))) p *
    field_at Tsh t.struct_tree [StructField .value] v p *
    treebox_rep l (field_address t.struct_tree [StructField .left] p) *
    treebox_rep r (field_address t.struct_tree [StructField .right] p)
end.

Here, val is CompCert Clight’s value type. “&&”, “*” and “EX” are notations for conjunction, separating conjunction and existential quantifiers in Verifiable C’s unlifted assertion language. “!!” is the notation that injects Coq propositions into
the unlifted assertion language. The expression \((\text{Vint (\text{Int.repr (Z.of_nat x)})})\) injects a natural number \(x\) into the integers, then to a 32-bit integer\(^5\) then to CompCert Clight’s value type, \(\text{val}\).

### 10.3.2 Example: partial trees

In section 7.2, I claim that the soundness of rules (7.2c) (7.2d) and (7.2e) does not depend on the definition of \(\text{TBrep(\_\_)}\). Here are those three rules:

\[
\Sigma; \Gamma; \text{emp} \vdash \text{PTBrep}(p, p, \lambda t. t)
\]
\[
\Sigma; \Gamma; \text{TBrep}(i, t) \ast \text{PTBrep}(r, i, P) \vdash \text{TBrep}(r, P(t))
\]
\[
\Sigma; \Gamma; \text{PTBrep}(p_1, p_2, P_1) \ast \text{PTBrep}(p_2, p_3, P_2) \vdash \text{PTBrep}(p_1, p_3, P_1 \circ P_2)
\]

And here are their counterparts for \(\text{Trep(\_\_)}\) (rules (7.3c) (7.3d) and (7.3e)):

\[
\Sigma; \Gamma; \text{emp} \vdash \text{PTrep}(p, p, \lambda t. t)
\]
\[
\Sigma; \Gamma; \text{Trep}(i, t) \ast \text{PTrep}(r, i, P) \vdash \text{Trep}(r, P(t))
\]
\[
\Sigma; \Gamma; \text{PTrep}(p_1, p_2, P_1) \ast \text{PTrep}(p_2, p_3, P_2) \vdash \text{PTrep}(p_1, p_3, P_1 \circ P_2)
\]

Formally specifying these proof rules in Coq enables us to clearly convey the meaning of “not depend on the definition of \(\text{TBrep(\_\_)}\)”.

I first define \texttt{partialT} which is a partial tree predicate parameterized over complete-tree representation predicates. And then \(\text{PTBrep(\_\_\_)}\) (called \texttt{partial_treebox_rep} in Coq) and \(\text{PTrep(\_\_\_)}\) (called \texttt{partial_tree_rep} in Coq) are just instances of it.

\(^{5}\)Mapping \(\mathbb{Z}\) to \(\mathbb{Z}\) mod \(2^{32}\) is not injective; in a practical application the client of this search-tree module should prove that \(x < 2^{32}\).
Definition partialT (rep: tree val → val → pred heap)
(P: tree val → tree val) (p.root p.intern: val) :=

ALL t: tree val, rep t p.intern -* rep (P t) p.root.

Definition partial_treebox_rep := partialT treebox_rep.
Definition partial_tree_rep := partialT tree_rep.

After that, I prove the following parameterized proof rules sound. For example, rule (7.2d) and rule (7.3d) are two instances of rep_partialT_rep. Figure 10.3 illustrates this common pattern beyond these two rules.

The soundness proofs of these parameterized rules use wandQ-frame rules. I only list one proof of these three here for the sake of space.

Lemma emp partialT rep H: forall rep p,

emp ⊢ partialT rep (fun t => t) p p.
Proof. ... Qed.

Lemma rep partialT rep: forall t P p q, rep t p * partialT rep P q p ⊢ rep (P t) q.
Proof. intros. exact (wandQ.frame_elim _ (fun t => rep t p) (fun t => rep (P t) q) t). Qed.

Lemma partialT rep partialT rep: forall rep pt12 pt23 p1 p2 p3,

partialT rep pt12 p2 p1 * partialT rep pt23 p3 p2 ⊢
partialT rep (Basics.compose pt23 pt12) p3 p1.
Proof. ... Qed.

Figure 10.3: Diagrams of rule (7.2d) and rule (7.3d).
Summary. VST-Floyd both provides predicates for C types (\texttt{data.at} and \texttt{field.at}) and allows users to define their domain-specific predicates. VST-Floyd proves the separation logic proof theory for C types sound. Users need to prove their own domain-specific proof rules.
Chapter 11

Canonical form

In this chapter, I introduce the canonical form of assertions which plays an important role in VST-Floyd. In the past six years, this canonical form has been continuously improved by new insights from program verification practice. Now, with this canonical form, users can write assertions succinctly and VST-Floyd can efficiently perform forward symbolic execution (see Chapter 12 and 13). More specifically, C program expression evaluation can be computationally derived from a canonical assertion (section 11.4); program variable substitution on canonical assertions are symbolic (section 11.5). Andrew W. Appel designed the first prototype of this canonical form. He and I together contributed to later improvements and reformulation.

Our canonical form segregates a separation-logic assertion (a predicate over stack-heap pairs) into three parts (interpreted conjunctively):

PROP: pure propositions that are independent of stack and heaps
LOCAL: values of nonaddressable variables and addresses of addressable variables
SEP: spatial separation-logic predicates that are independent of stacks.
Formally, a canonical assertion has the form

\[ \text{PROP}(P_1; P_2; \ldots) \text{LOCAL}(Q_1; Q_2; \ldots) \text{SEP}(R_1; R_2; \ldots) \]

The \( P_i \) have type \( \text{Prop} \) in Coq. The \( Q_i \) are fully syntactic (deeply embedded), and have denotations in type \( \text{stack} \to \text{Prop} \); and the \( R_i \) are predicates over heaps (i.e. assertions in the unlifted separation logic).

Every conjunct in the LOCAL part is one of the following:

- \text{temp} \; x \; v, \quad \text{meaning that } x \text{ is a nonaddressable variable and } [x] = v
- \text{lvar} \; x \; v, \quad \text{meaning that } x \text{ is a local addressable variable and } [\&x] = v
- \text{gvar} \; x \; v, \quad \text{meaning that } x \text{ is a global variable and } [\&x] = v.

For example, this assertion in canonical form

\[ \text{PROP}(a \geq 0; b \geq 0) \; \text{LOCAL(temp} \; p) \; \text{SEP}(p \mapsto a; (p + 4) \mapsto b) \]

represents:

\[ a \geq 0 \land b \geq 0 \land [x] = p \land p \mapsto a \land (p + 4) \mapsto b. \]

A useful property of this canonical form is that the PROP and SEP parts are independent of the stack. This correspond to the program-variable-isolated form used previously in this thesis. Their conjuncts cannot test the value of nonaddressable variables or the address of addressable variables directly; they must do so indirectly, using auxiliary variables (i.e. Coq variables) shared with the LOCAL part. In the example above, all communication between the C variable \( x \) and its properties is done by means of Coq variable \( p \).

Other than this restriction, the canonical form is flexible in its PROP and SEP parts. In the rest of this chapter I demonstrate some examples of triples (section \[11.1\]) and function specifications (section \[11.2\]) in canonical form. I discuss the expressiveness

---

\(^1\)In the Coq development, we use the name environ for what I call “stack” in the thesis.
of canonical assertions (section 11.3). And I will introduce computational derivation of C expression evaluation and symbolic C variable substitution which are both based on canonical assertions.

11.1 Examples: triples of atomic C commands

The following is a program fragment, to demonstrate Hoare triples of set, load and store commands. The assertions in this decorated program are all written in canonical form, and the postconditions are strongest postconditions.

```c
int *x; int y, z;

{PROP() LOCAL(temp x; temp y a) SEP(p \rightarrow int b; p + 4 \rightarrow int 0)}
   z = *x;
{PROP() LOCAL(temp x; temp y a; temp z b) SEP(p \rightarrow int b; p + 4 \rightarrow int 0)}
   y = y + z;
{PROP() LOCAL(temp x; temp y (a + b); temp z b) SEP(p \rightarrow int b; p + 4 \rightarrow int 0)}
{PROP() LOCAL(temp x; temp y (a + b)) SEP(p \rightarrow int b; p + 4 \rightarrow int 0)}
   x = x + 1;
{PROP() LOCAL(temp x (p + 4); temp y (a + b)) SEP(p \rightarrow int b; p + 4 \rightarrow int 0)}
   *x = y;
{PROP() LOCAL(temp x (p + 4); temp y (a + b)) SEP(p \rightarrow int b; p + 4 \rightarrow int a + b)}
```

11.2 Examples: function specifications

Suppose we have a C function with the following signature:
\begin{verbatim}
struct IntPair { int fst; int snd; };
void swapIntpair(struct IntPair * x);
\end{verbatim}

where \texttt{IntPair} is from an example in section 10.2 and \texttt{swapIntPair} swaps the numbers stored in the two fields.

Naturally, its specification can be described as follows:

\[
\text{swapIntpair} : \text{WITH } a, b, p.
\]
\[
\begin{align*}
\text{PRE:} & \quad [x] = p \land p \xrightarrow{\text{IntPair}} a, b \\
\text{POST:} & \quad p \xrightarrow{\text{IntPair}} b, a
\end{align*}
\]

It is worth noticing that isolating program variables in the precondition is necessary here because we want to claim that the swapped pair of numbers are stored at the same address as before.

In VST-Floyd, this specification is described in the following canonical form:

\[
\text{swapIntpair} : \text{WITH } a, b, p.
\]
\[
\begin{align*}
\text{PRE:} & \quad \text{PROP}() \text{LOCAL}(\text{x} \times p) \text{SEP}(p \xrightarrow{\text{IntPair}} a, b) \\
\text{POST:} & \quad \text{PROP}() \text{LOCAL}() \text{SEP}(p \xrightarrow{\text{IntPair}} b, a)
\end{align*}
\]

Sometimes it is useful to wrap an existential quantifier around a canonical-form assertion in the postcondition, for example:
void sort(int * bg; int * ed);

sort : WITH l, p, n.

PRE: PROP()
    LOCAL(temp bg p; temp ed (p + 4n))
    SEP(p \mapsto l)

POST: \exists l'. PROP(Permutation(l, l'); ordered(l'))
    LOCAL()
    SEP(p \mapsto l')

User-defined predicates in the unlifted separation logic can describe data structures in memory. I have defined TBrep(_, _) in Coq in section 10.3. Using that Coq defined separation logic predicate, the BST-insert’s implementation correctness (see section 7.1) can be specified as follows in VST-Floyd:

insert : WITH p0, t0, x, v.

PRE: PROP()
    LOCAL(temp p p0; temp x x; temp v v)
    SEP(TBrep(p0, t0))

POST: PROP()
    LOCAL()
    SEP(TBrep(p0, ins(t0, x, v)))

11.3 Discussion: expressiveness

In the examples that I presented above (especially the triples in section 11.1), we have seen the convenience achieved by limiting the use of C variables in canonical
assertions. Specifically, the postcondition of a set command or a load command only modifies or adds one conjunct in the \textsc{local} clauses. With a store command, the postcondition’s \textsc{sep} clause differs only in one term from the precondition’s. It is natural to ask whether this setting restricts the expressiveness of the assertion language. I answer this question here.

Observation one: any assertion can be decomposed into existentially quantified canonical form. Let $P$ be any assertion about (nonaddressable) local variables $x_1, x_2, \ldots, x_n$. Then $P$ can be decomposed into,

$$\exists x_1 x_2 \ldots x_n.$$ 

$$\text{PROP()} \ \text{LOCAL}(\text{temp } x_1; \ldots; \text{temp } x_n \ \text{SEP}(P[x_1/x_1, \ldots, x_n/x_n])$$

Addressable local and global variables can be substituted using \textsc{lvar} and \textsc{gvar}.

Observation two: existential quantifiers in preconditions can be eliminated using the Hoare rule,

$$\text{HOARE-EXISTS:} \quad x \text{ does not freely occur in } \Gamma, Q \text{ or } \vec{R}$$

$$\text{then } \Sigma; \Gamma; \Delta \vdash \{ \exists x. P \} \ c \ \{ Q, [\vec{R}] \}.$$  

When doing proof by forward symbolic execution, one applies this rule and exposes the underlying canonical-form precondition $P$ for further manipulation.

To conclude, VST-Floyd requires all preconditions in function specifications to be in canonical form and requires all postconditions to be existentially quantified canonical assertions. This setting does not decrease expressiveness and it is actually very practical for describing the behavior of C functions.
11.4 Computational derivation of expression evaluation

Given a primary r-value expression \( e \) (i.e. computing \( e \)'s denotation is memory irrelevant) and a canonical precondition \( \text{PROP}() \text{ LOCAL } \overrightarrow{Q} \text{ SEP } \overrightarrow{R} \), sometimes (e.g. when generating the strongest postconditions for set, load or store commands, see chapter 12) we need to find a value \( v \) such that

\[
\text{PROP}() \text{ LOCAL } \overrightarrow{Q} \text{ SEP } \overrightarrow{R} \vdash [e] = v
\]

Without the canonical form, this task is complicated, even if the precondition is as simple as in the following example:

\[
[x] = 1 \land [y] = 1 \land [z] = 0 \vdash [x+y] = v
\]

In this example, we can reduce this task to the following Coq proof goal:

\[
\begin{align*}
\text{s: stack} \\
\text{H: } [x] s &= 1 \\
\text{H0: } [y] s &= 1 \\
\text{H1: } [z] s &= 0 \\
\hline
\text{-------------------------} \\
[x] s + [y] s &= ?v
\end{align*}
\]

Then, we can look for useful assumptions and rewrite them in the conclusion, i.e. \( \text{rewrite H, H0} \). However, searching useful assumptions needs careful tactic programming. Moreover, both proof search and rewriting are slow in Coq.
We define two mutually recursive functions \texttt{msubst\_eval\_expr} (for evaluating r-value expressions) and \texttt{msubst\_eval\_lvalue} (for evaluating the address of l-value expressions) which do symbolic evaluation independent of stack\footnote{In our Coq development, we actually turn the symbolic \texttt{LOCAL} clauses into binary trees first. Then we look up in these trees during symbolic evaluation. We omit the technical details here.}:

\begin{align*}
\texttt{msubst\_eval\_expr} & : \texttt{Clight.expr} \rightarrow \texttt{list localdef} \rightarrow \text{option val} \\
\texttt{msubst\_eval\_lvalue} & : \texttt{Clight.expr} \rightarrow \texttt{list localdef} \rightarrow \text{option val}
\end{align*}

Their definitions are almost the same as \([e]\) and \([&e]\). The only difference is that \([e]\) and \([&e]\) look up values of nonaddressable variables and addresses of addressable variables in stacks but \texttt{msubst\_eval\_expr} and \texttt{msubst\_eval\_lvalue} look them up in local clauses. We prove the following lemmas and use them to construct expression evaluation’s results directly.

\begin{align*}
\text{MSUBST-EXPR} : \ & \Sigma; \Gamma; \ \text{msubst\_eval\_expr}(e, \vec{Q}) = \text{Some}(v); \\
& \quad \text{PROP}() \ \text{LOCAL} \quad \vec{Q} \ \text{SEP} \quad \vec{R} \vdash [e] = v \\
\text{MSUBST-LVALUE} : \ & \Sigma; \Gamma; \ \text{msubst\_eval\_lvalue}(e, \vec{Q}) = \text{Some}(v); \\
& \quad \text{PROP}() \ \text{LOCAL} \quad \vec{Q} \ \text{SEP} \quad \vec{R} \vdash [&e] = v
\end{align*}

In VST-Floyd, it is typical that we need to generate a Coq value \(v\) (that means \(v\) is an expression independent of program state) such that:

\begin{align*}
\Sigma; \Gamma; \ \text{PROP}() \ \text{LOCAL} \quad \vec{Q} \ \text{SEP} \quad \vec{R} \vdash [e] = v.
\end{align*}

By \texttt{MSUBST-LVALUE}, we only need to prove the following Coq proof goal and instantiate the Coq unification variable (\(?v\)) in it:

\begin{align*}
\texttt{msubst\_eval\_expr}(e, \vec{Q}) = \text{Some} \ ?v
\end{align*}

\footnote{In our Coq development, we actually turn the symbolic \texttt{LOCAL} clauses into binary trees first. Then we look up in these trees during symbolic evaluation. We omit the technical details here.}
This can be done simply by Coq’s \textit{reflexivity}. We build the following tactic \texttt{solve.msubst.eval.expr} which not only instantiates this unification variable \(?v\), but also simplifies its expression. For example, if \(p\) is known to be a pointer, than “\texttt{offset.val 0 p}” will be replaced by \(p\). Besides, an analogous tactic \texttt{solve.msubst.eval.lvalue} is built for l-values.

\begin{verbatim}
Ltac solve.msubst.eval.expr :=
  simpl;
  cbv beta iota zeta delta [force_val2 force_val1];
  rewrite ?isptr.force.ptr, < ?offset_val.force.ptr by auto;
  reflexivity.
\end{verbatim}

### 11.5 Symbolic program variable substitution

If an assignment command modifies a nonaddressable variable (e.g. a set command or a load command), its forward rule (Robert W. Floyd’s assignment rule, to distinguish from C. A. R. Hoare’s “backward” assignment rule) contains an existential quantifier and variable substitution.

\begin{align*}
\text{Floyd} : \{P\} & \ x := E \{\exists y. x = E[y/x] \land P[y/x]\} \\
\text{Hoare} : \{Q[E/x]\} & \ x := E \{Q\}
\end{align*}

Quantifiers and variable substitutions make proofs inconvenient, especially interactive proofs. Actually, this is one of the most important drawbacks of forward verification, compared to backward verification. Fortunately, variable substitution on canonical assertions can be reduced away. Specifically, if \(x\) is a nonaddressable
variable and $v$ is a program-variable-irrelevant expression, then:

\[
\Sigma; \Gamma; \ (\text{temp} \times v) \in \vec{Q};
\]

\[
(\text{PROP } \vec{P} \ \text{LOCAL } \vec{Q} \ \text{SEP } \vec{R})[x_0/x] \vdash
\]

\[
\text{PROP}(x_0 = v; \vec{P})\text{LOCAL}(\text{remove_temp}(x, \vec{Q}))\text{SEP } \vec{R}
\]

\[
\Sigma; \Gamma; \ (\text{temp} \times \_\text{)} \ does \ not \ appear \ in \ \vec{Q};
\]

\[
(\text{PROP } \vec{P} \ \text{LOCAL } \vec{Q} \ \text{SEP } \vec{R})[x_0/x] \dashv\vdash
\]

\[
\text{PROP } \vec{P} \ \text{LOCAL } \vec{Q} \ \text{SEP } \vec{R}
\]

Here, \texttt{remove\_temp} is a Coq function which deletes conjuncts about a specific nonaddressable from the local clauses (if there exists one). This rule is sound because: (1) both \texttt{PROP} and \texttt{SEP} parts are independent of stacks; thus variable substitutions have no effect on them, and (2) all \texttt{LOCAL} conjuncts other than “temp $x$” are not affected by substituting $x$.

Actually, in the first lemma above, the left side and the right are equivalent (instead of left implies right), if $x$ does not appear multiple times in the \texttt{LOCAL} part. Later, we will use the direction from left to right to prove soundness of some derived Hoare logic rules while the direction from right to left ensures that the generated postconditions are strongest postconditions. We do not prove this “strongest” property in Coq. Thus, we do not need to prove right-to-left in Coq either.

As an immediate consequence of two lemmas above, existentially quantified old values can be eliminated together with substitution, i.e.

\[
\Sigma; \Gamma; \ \exists x_0. (\text{PROP } \vec{P} \ \text{LOCAL } \vec{Q} \ \text{SEP } \vec{R})[x_0/x] \vdash
\]

\[
\text{PROP } \vec{P} \ \text{LOCAL}(\text{remove_temp}(x, \vec{Q}))\text{SEP } \vec{R}
\]

This is easily proved sound by case analysis.
11.6 Other canonical form in previous works

Berdine et al. [12] first proposed a canonical form of separation logic which distinguishes pure facts and spatial facts. Specifically, an assertion of their canonical form can be represented as \((P_1 \land \cdots \land P_n) \land (Q_1 \ast \cdots \ast Q_m)\) in which \(P_i\) are nonspatial and \(Q_i\) are spatial. They did not isolate program variables as we do.

Charge! [9] is a separation logic tool proved sound in Coq. Like ours, their assertions are predicates over stack-heap pairs. In comparison, the separating conjuncts in their “canonical form” of assertions are not required to be independent of program variables. As a consequence, generating a strongest postcondition is more complicated.

Iris Proof Mode (IPM) [38] has an assertion form \(\bigstar P_i \ast \bigstar L_i \rightarrow \bigstar R_i\) where \(P_i\) are the persistent conjuncts. Based on this canonical form, IPM implements a proof mode (IPM) inside another proof mode (Coq). Specifically, IPM provides tactics iApply, iIntro etc, which work like Coq’s built-sin apply and intro. Because IPM is designed for ML-like languages whose local variables are substitution-based, IPM naturally lacks automation for Algol-style (C-style) local variables.
Chapter 12

Generating strongest
postconditions for assignment
commands

In section 11.1, I illustrated how assignment commands can be concisely characterized
by Hoare triples with canonical assertions. Now I demonstrate how those postcondi-
tions in triples can be generated. VST-Floyd produces strongest postconditions, with
soundness proofs for the corresponding triples. The property of being strongest post-
conditions is not proved in Coq; it is only a metaproperty ensuring that any sound
Hoare triple can be proved by VST-Floyd.

Formally speaking, when the command $c$ is a primary assignment command (a
set, load, store, or function call), our tactics in Coq prove the following judgment,
instantiating the unification variable $?Post$ with a strongest postcondition.

$$
\Sigma; \Gamma; \Delta \vdash \left\{ \text{PROP}() \ \text{LOCAL} \ Q \ \text{SEP} \ R \right\} \ c \ \left\{ ?\text{Post}, [\mathbb{I}] \right\}
$$
Because an assignment statement or function call does not \textbf{break}, \textbf{continue}, or \textbf{return}, those three postconditions can be false. Only the postcondition for normal termination is nontrivial.

We can assume that every C variable appears at most once in $\overrightarrow{Q}$. Otherwise, e.g. if $\overrightarrow{Q}$ were $[\text{temp} \times x_1; \text{temp} \times x_2]$, we could remove a conjunct and add $x_1 = x_2$ into the PROP part.

We can also assume canonical preconditions have empty PROP clauses because they can all be moved “above the line” into Coq assumptions of the whole triple. Still, PROP part is useful in other assertions like existentially quantified canonical assertions and precondition in parameterized function specifications.

To solve such Coq proof goals, the main task is to build Coq tactics that apply Hoare logic proof rules. Verifiable C provides elementary proof rules for set, load, store and function call. However, they are not easy to use. Thus,

1. we derive new proof rules which have canonical preconditions,

2. we organize the assumptions of these proof rules carefully so that solving them in order directly instantiates the postcondition, and

3. we use \textit{domain-specific small scale reflection} to implement some complicated proof automation.

\section*{12.1 Set command}

I first introduce strongest postcondition derivation for set commands. Recall that a set command in C is an assignment command whose left side is a nonaddressable variable and right side is a primary r-value expression (see section \ref{sec:eval}). Based on previous theorems of C expression evaluation and symbolic variable substitution, we
prove the following derived rule from SEMAX-SET (see section 2.6.3):

SEMAX-SET-CANON:

If \( \Sigma; \Gamma; \text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R} \vdash e \downarrow \) and
\( \Sigma; \Gamma \vdash \text{msubst\_eval\_expr}(e, \overrightarrow{Q}) = \text{Some}(v), \)
then \( \Sigma; \Gamma; \Delta \vdash \{ \text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R} \} \)
\( x = e \)
\( \{ \text{PROP()} \text{ LOCAL} (\text{temp\_x} \cdot v; \text{remove\_temp}(x, \overrightarrow{Q})) \text{ SEP} \overrightarrow{R}, [\overrightarrow{\bot}] \} \)

It is formalized in Coq as follows:

**Lemma SEMAX-SET-CANON:**

forall \( \Delta \overrightarrow{Q} \overrightarrow{R} x e v, \)
\( \text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R} \vdash \text{tc\_expr} \Delta e \rightarrow \)
\( \text{msubst\_eval\_expr} e \overrightarrow{Q} = \text{Some} v \rightarrow \)
\( \text{semax} \Delta (\text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R}) \)
\( (x = e) \)
\( (\text{PROP()} \text{ LOCAL} (\text{temp\_x} \cdot v; \text{remove\_temp}(x, \overrightarrow{Q})) \text{ SEP} \overrightarrow{R}, [\overrightarrow{\bot}] \). \)

Here, “\( \text{tc\_expr} \Delta e \)” is Verifiable C’s formalization of \( e \downarrow \). Similarly, “\( \text{tc\_value} \Delta e \)” is Verifiable C’s formalization of \( &e \downarrow \).

To prove the following judgment (\( e \) is a primary r-value expression):

\( \Sigma; \Gamma; \Delta \vdash \{ \text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R} \} \ x = e \ \{ ?\text{Post}, [\overrightarrow{\bot}] \} \)

or equivalently, to solve the following Coq proof goal:

\[ \text{semax} \Delta (\text{PROP()} \text{ LOCAL} \overrightarrow{Q} \text{ SEP} \overrightarrow{R}) \ (x = e) \ \{ ?\text{Post}, [\overrightarrow{\bot}] \} \]

---

1 All proof rules for assignment commands also need to check whether the C type on the left side is compatible with the C type on the right side. I omit them for brevity.
we run “\texttt{eapply semax\_set\_canon}” in Coq. Two proof goals will be left. One proof goal is the typechecking requirement. Most of the time, it can be solved automatically, i.e. the right side is just true. When \( e \) is an expression like \( y/z \), then \( \text{tc\_expr}(\Delta, e) \) computes to the assertion \([z] \neq 0\), which the user must prove manually (if the tactic automation does not solve it automatically).

The other proof goal is:

\[ ...
\]
\[ ......................................
\]
\[ \text{msubst\_eval\_expr } e \overset{Q}{\Rightarrow} = \text{Some } ?v
\]

We use “\texttt{solve\_msubst\_eval\_expr}” (see section 11.4) to solve this proof goal and fill the unification variable in it.

By all these steps above, together with the derived Hoare rule for set commands, we have already generated a (strongest) postcondition of the required set command and a soundness proof of corresponding triple. The generated postcondition is in canonical form. Its \textsc{prop} part is empty; its \textsc{sep} part is the same as in the precondition; the \textsc{local} part replaces the conjunct of “\texttt{temp x}” with the expression evaluation result.

### 12.2 Load command and store command

The previous section introduces strongest postcondition generation for set commands. Set commands are heap-independent. This sections turns to heap-related commands—load commands and store commands (see section 2.1). Recall that a \textit{load} command is an assignment whose left side is a nonaddressable variable and right side is a primary l-value. A \textit{store} command is an assignment whose left side is a primary l-value and whose right side is a primary r-value.
The effect of load is first computing the loaded address and then using the value stored in that address in the heap to update a C variable. Given a precondition and a C load command, the goal of strongest postcondition generation is thus to find the SEP clause which ought to be loaded from, to compute the loaded value and to modify the LOCAL clause.

The effect of store, on the other hand, is first computing the assigned value and the stored address then updating the heap accordingly. Thus, generating the strongest postcondition needs to find the SEP clause affected by the store and to modify it with the assigned value.

The following is an example of load and store:

```c
struct IntPair { int fst; int snd; }
struct IntPair *x;
int t;

{PROP()LOCAL(temp x)SEP(x \mapsto \rightarrow IntPair a, b)}
t = x \rightarrow \rightarrow fst;
{PROP()LOCAL(temp t a; temp x)SEP(x \mapsto \rightarrow IntPair a, b)}
x \rightarrow \rightarrow snd = 0;
{PROP()LOCAL(temp t a; temp x)SEP(x \mapsto \rightarrow IntPair a, 0)}
```

Figure 12.1: Strongest Postcondition Generation Examples

The process of generating the strongest postconditions for load commands and store are similar. Both of them need to compute the load/store address and to find the related SEP clause. So I describe them together in one section.

**Design choice.** In the strongest postconditions generation tactics, VST-Floyd only handles `data.at` and `field.at` in SEP clauses. On one hand, users can define any spatial predicates for their domain-specific demand and it is impossible to cover all of them. On the other hand, this is a safe designed choice—user-defined predicates, like `TBrep(_, _)`, can be unfolded so that `data.at` or `field.at` is exposed.
12.2.1 Derived proof rules

I first pose the derived rules here and then explain them step by step in later subsections.

**Lemma SEMAX_LOAD_CANON**: for all $n \ Rn \ \Delta \ \overrightarrow{Q} \ \overrightarrow{R} \times e$

$e_r \overrightarrow{F} \ lr \ t_r \overrightarrow{f} \ p \ t'_r \overrightarrow{f}_0' \overrightarrow{f}_1' \ p' \ v \ v'$,
compute_nested_efield $e = (e_r, \overrightarrow{F}, lr) \rightarrow$
msubst_eval_LR $e_r \overrightarrow{Q} \ lr = Some \ p \rightarrow$
msubst_efield_denote $\overrightarrow{F} \ \overrightarrow{Q} \ \overrightarrow{f} \rightarrow$
compute_root_type (typeof $e_r) \ lr \ t_r \rightarrow$
field_address_gen $(t_r, \overrightarrow{f}, p) (t'_r, \overrightarrow{f}', p') \rightarrow$
find_nth_preds
(fun $Rn \Rightarrow Rn = field\_at \ t'_r \overrightarrow{f}_0' \overrightarrow{v'} \ p' \ \wedge \overrightarrow{f}' = \overrightarrow{f}_0' \cdot \overrightarrow{f}_1'$)
$\overrightarrow{R} (Some (n, Rn)) \rightarrow$
JMeq $(v'.\overrightarrow{f}_1') v \rightarrow$
PROP () LOCAL $\overrightarrow{Q} SEP \overrightarrow{R} \vdash!! (legal\_nested\_field \ (t'_r.\overrightarrow{f}_0') \overrightarrow{f}_1') \rightarrow$
PROP () LOCAL $\overrightarrow{Q} SEP \overrightarrow{R} \vdash (tc\_LR \ \Delta \ e_r \ lr) \&\& (tc\_efield \ \Delta \ \overrightarrow{F}) \rightarrow$
semax $\Delta (PROP () LOCAL \overrightarrow{Q} SEP \overrightarrow{R})$
$(x = e)$

(PROP () LOCAL (temp $x \ v :: remove\_localdef\_temp \times \overrightarrow{Q}) SEP \overrightarrow{R}, [\overrightarrow{\bot}]).
Lemma SEMAX_STORE_CANON: forall n Rn ∆ Q R e1 e2

\[ e_r \xrightarrow{F} lr t_r \xrightarrow{f} p t'_r \xrightarrow{f'_0} f'_1 p' u' u' v Rv, \]
compute_nested_efield \( e_1 = (e_r, \xrightarrow{F}, lr) \rightarrow \)
msubst_eval_expr \( e_2 = \text{Some } u' \rightarrow \)
msubst_eval_LR \( e_r, lr = \text{Some } p \rightarrow \)
msubst_efield_denote \( F \xrightarrow{Q} f \rightarrow \)
compute_root_type (typeof \( e_r \)) lr t_r \rightarrow
field_address_gen \( (t_r, \xrightarrow{f}, p) (t'_r, \xrightarrow{f'}, p') \rightarrow \)
find_nth_preds
\[
\begin{align*}
& \quad \text{fun } Rn \Rightarrow (Rn = Rv v' \land (Rv = \text{fun } v \Rightarrow \text{field.at } t'_r \xrightarrow{f'_0} v p')) \land f' = f'_0 \cdot f'_1) \\
& \quad \xrightarrow{R} (\text{Some } (n, Rn)) \rightarrow \\
& \quad \text{JMeq } u' u \rightarrow \\
& \quad \text{JMeq } \left( v' \left[ f'_1 \mapsto u \right] \right) v \rightarrow \\
& \quad \text{PROP } () \text{ LOCAL } Q \text{ SEP } R \vdash !! \left( \text{legal_nested_field } \left( t'_r. f'_0 \right) f'_1 \right) \rightarrow \\
& \quad \text{PROP } () \text{ LOCAL } Q \text{ SEP } R \vdash (\text{tc.LR } e_r, lr) \land (\text{tc.expr } e_2) \land (\text{tc.efield } f) \rightarrow \\
& \text{semax } \Delta (\text{PROP } () \text{ LOCAL } Q \text{ SEP } R) \rightarrow \\
& \quad (e_1 = e_2) \\
& \quad (\text{PROP } () \text{ LOCAL } Q \text{ SEP } (\text{replace_nth } n R (Rv v)), [\overline{l}]).
\end{align*}
\]

These two derived rules are proved in favor of proof automation, rather than for an elegant proof theory. Here is the Coq proof goal corresponding to strongest postcondition generation task for the load command in Figure 12.1. I use it to introduce the brief process of strongest postcondition generation here.
\( \Delta := \ldots \)

\[
\text{semax} \ \Delta \ \text{(PROP()) LOCAL(temp x x) SEP(x} \xrightarrow{\text{IntPair}} a, b))
\]

\[
(t = x \rightarrow \text{fst};)
\]

\[
(?\text{Post}, [\perp])
\]

The strongest postcondition generator, which is an Ltac program in Coq, will first \textit{eapply SEMAX-LOAD-CANON}. The variables like \( \Delta, \overrightarrow{Q} \) and \( e \) in \texttt{SEMAX-LOAD-CANON} are instantiated from the proof goals. For example, \( e \) is instantiated by \( x \rightarrow \text{fst} \). How to instantiate others variables like \( e_r \) and \( p \) is not known yet. They will be represented by new Coq unification variables and are supposed to be filled later. This \textit{eapply} turns the original proof goal into 9 subgoals, which correspond to 9 assumptions of \texttt{SEMAX-LOAD-CANON}. For example, the first proof goal is:

\[
\ldots
\]

\[
\text{compute_nested_efield} \ x \rightarrow \text{fst} = (?e_r, ?\overrightarrow{F}, ?l_r)
\]

I build Coq tactics to solve these proof goals one by one—unification variables are filled one by one in this process. In the end, all these 9 proof goals are solved and all unification variables are instantiated. That means, the original proof goal is proved and the postcondition \( ?\text{Post} \) is filled.

\subsection{12.2.2 Computing syntactic path}

The first step of postcondition generation for load/store commands is to figure out the loaded/stored address. In Figure [12.1] the loaded address is \( x.\text{fst} \) and the stored address is \( x.\text{snd} \). That means: we should look for a SEP clause with form "\texttt{data.at \texttt{IntPair} - x}" (or in short \( x \xrightarrow{\text{IntPair}} . \)).

124
In these two examples, addresses \(x.fst\) and \(x snd\) directly correspond to the primary l-value expressions, \(x \rightarrow fst\) and \(x \rightarrow snd\) here. Such situations are common. Thus the very first step here is to destruct a primary l-value expression into a root expression and a syntactic path. The Coq function \(\text{compute\_nested\_efield}\) does this job.

A syntactic path of nested fields has Coq type \(\text{list\ efield}\). The Coq type \(\text{efield}\) is similar to, but different from \(\text{gfield}\). The difference is, in order to represent an array subscript, \(\text{efield}\) uses a C expression (which should be a primary r-value expression with integer type) but \(\text{gfield}\) uses an integer. In other words, \(\text{efield}\) is syntactic but \(\text{gfield}\) is denotational.

It is tricky that a root expression can be either a primary l-value expression or a primary r-value expression. In the example above, \(x \rightarrow fst\) is actually \((\ast x).fst\). Here, \(\ast x\) is the root expression and is a primary l-value expression. But in a pointer-as-array situation, the root expression will be a primary r-value expression. For example, consider

```bash
int a[10], * b, i;

b = a + 5;

b[i] = 7;
```

Here, \(b[i]\) is a primary l-value expression. In Clight, an array subscript is treated as syntactic sugar for addition and dereference. Thus, \(b[i]\) is actually \((b + i)\) and \(b\), a primary r-value expression, is the root expression here.

The Coq function \(\text{compute\_nested\_efield}\) takes care of all these different situations. It computes the syntactic path, the root expression and decides whether it is an l-
value or a r-value. For example, “b[i]” is split into (b, [i], R) and “x → fst” is split into (x, [fst], L). Fields and array subscripts can be nested. For example, in the following C program:

```c
struct IntPair a[10];

a[2].fst = 0;
```

The expression “a[2].fst” is split into (a, [2; fst], R).

In Coq, I define the follow Coq type for indicating l-value or r-value (in short, L and R).

```coq
Inductive LLRR : Type :=
| LLLL : LLRR
| RRRR : LLRR.
```

The Coq definition of `compute_nested_efield` is long; it can be found in file floyd/e-field_lemmas.v. The following is its type information

```coq
Definition compute_nested_efield (e: expr): expr * list efield * LLRR := ...
```

12.2.3 Computing load/store addresses

After computing the syntactic path, we need to turn it into an address in a form of “field_address $t_r \rightarrow f p$”. We enforce this field_address form because this can be used as a clue to which SEP clause we are looking for. The following table shows some examples.
<table>
<thead>
<tr>
<th>Primary l-value expressions</th>
<th>Syntactic paths</th>
<th>LOCAL clauses</th>
<th>Load/store addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[i]</td>
<td>(b, [i], R)</td>
<td>[temp b b; temp i i]</td>
<td>field.address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(int[?n]) [i] b</td>
</tr>
<tr>
<td>x → fst</td>
<td>(*x, [fst], L)</td>
<td>[temp x x]</td>
<td>field.address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IntPair [fst] x</td>
</tr>
<tr>
<td>a[2].fst</td>
<td>(a, [2; fst], R)</td>
<td>[temp a p]</td>
<td>field.address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(IntPair[10]) [2; fst] p</td>
</tr>
</tbody>
</table>

Three assumptions in `SEMAX-LOAD-CANON/SEMAX-STORE-CANON` compute these three values \((t_r, \overrightarrow{f} \text{ and } p \text{ in } \text{“field.address } t_r \overrightarrow{f} \text{ p”})\) respectively:

\[
\begin{align*}
\text{msubst.eval.LR } e_r \overrightarrow{Q} \text{ lr} &= \text{Some } p \\
\text{msubst.efield.denote } \overrightarrow{\text{f}} \overrightarrow{Q} \overrightarrow{f} \\
\text{compute_root_type } (\text{typeof } e_r) \text{ lr } t_r
\end{align*}
\]

**Computing root addresses**

The Coq function `msubst.eval.LR` computes the root address from the root expression. Its computation depends on whether the root expression is an l-value or a r-value.

**Definition** `msubst.eval.LR \overrightarrow{Q} e (lr: LLRR): option val :=`

```coq
match lr with
| LLLL ⇒ msubst.eval.lvalue \overrightarrow{Q} e
| RRRR ⇒ msubst.eval.expr \overrightarrow{Q} e
end.
```

127
Computing nested fields

The Coq predicate “\texttt{msubst\_efield\_denote} \overrightarrow{F} \overrightarrow{Q} \overrightarrow{f}” says that \( \overrightarrow{f} \) is the denotation of \( \overrightarrow{F} \) given the syntactic local clauses \( \overrightarrow{Q} \). \texttt{Msubst\_efield\_denote} is defined as a Coq inductive predicate and a tactic \texttt{solve\_msubst\_efield\_denote} is provided.

\textbf{Inductive} \texttt{Int\_eqm\_unsigned}: \texttt{int \rightarrow Z \rightarrow Prop} :=
| \texttt{Int\_eqm\_unsigned\_repr}: \forall z, \texttt{Int\_eqm\_unsigned} (\texttt{Int\_repr} z) z.

\textbf{Inductive} \texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \): \texttt{list efield \rightarrow list gfield \rightarrow Prop} :=
| \texttt{msubst\_efield\_denote\_nil}: \texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) nil nil
| \texttt{msubst\_efield\_denote\_cons\_array}: \forall ei i i' \overrightarrow{F} \overrightarrow{f}, \texttt{is\_int\_type} (\texttt{typeof} ei) = \texttt{true} \rightarrow
\texttt{msubst\_eval\_expr} \( \overrightarrow{Q} \) ei = \texttt{Some} (\texttt{Vint} i) \rightarrow
\texttt{Int\_eqm\_unsigned} i i' \rightarrow
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) \( \overrightarrow{F} \) \( \overrightarrow{f} \) \rightarrow
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) (\texttt{eArraySubsc} ei :: \( \overrightarrow{F} \)) (\texttt{ArraySubsc} i' :: \( \overrightarrow{f} \))
| \texttt{msubst\_efield\_denote\_cons\_struct}: \forall i \overrightarrow{F} \overrightarrow{f},
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) \( \overrightarrow{F} \) \( \overrightarrow{f} \) \rightarrow
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) (\texttt{eStructField} i :: \( \overrightarrow{F} \)) (\texttt{StructField} i :: \( \overrightarrow{f} \))
| \texttt{msubst\_efield\_denote\_cons\_union}: \forall i \overrightarrow{F} \overrightarrow{f},
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) \( \overrightarrow{F} \) \( \overrightarrow{f} \) \rightarrow
\texttt{msubst\_efield\_denote} \( \overrightarrow{Q} \) (\texttt{eUnionField} i :: \( \overrightarrow{F} \)) (\texttt{UnionField} i :: \( \overrightarrow{f} \)).
\textbf{Ltac} solve\_Int\_eqm\_unsigned := ...

\textbf{Ltac} solve\_msubst\_efield\_denote :=

\texttt{solute}

\[ \text{repeat first} \]

\[ \text{eapply msubst\_efield\_denote\_cons\_array;} \]

\[ \text{reflexivity} \]

\[ \text{| solve\_msubst\_eval\_expr} \]

\[ \text{| solve\_Int\_eqm\_unsigned} \]

\[ \text{|} \]

\[ \text{| apply msubst\_efield\_denote\_cons\_struct} \]

\[ \text{| apply msubst\_efield\_denote\_cons\_union} \]

\[ \text{| apply msubst\_efield\_denote\_nil} \]

\[ \text{|} \]

\[ \text{].} \]

\[ \text{].} \]

This combination of Coq inductive predicates and Coq tactics is a \textit{domain-specific small scale reflection}. The purpose is to generate $\vec{\mathcal{F}}$'s denotation $\vec{f}$. The inductive definition of \texttt{msubst\_efield\_denote} describes how this $\vec{f}$ can be generated: (1) when $\vec{\mathcal{F}}$ is empty, its denotation is also empty; (2) when the head of $\vec{\mathcal{F}}$ is an (syntactic) array subscript, the head of its denotation is the denotation of its head; (3) when the head of $\vec{\mathcal{F}}$ is a \texttt{struct/union} field, the head of its denotation is the same \texttt{struct/union} field.

The tactic \texttt{solve\_msubst\_efield\_denote} is designed to solve a proof goal with form “\texttt{msubst\_efield\_denote} $\vec{\mathcal{F}}$ $\vec{Q}$ "$\vec{f}$” in which the question mark represents an uninstantiated unification variable. It determines which constructor of \texttt{msubst\_efield\_denote} should be applied based on the shape of $\vec{\mathcal{F}}$. 

129
The array subscript case is the most interesting one. In the implementation of `solve_msubst_efield_denote`, `solve_msubst_eval_expr` is used to generate the denotation of syntactic array subscripts. The constructor `msubst_efield_denote_cons_array` requires the denotation to be a 32-bit integer (instead of other Clight values like pointers and floating point numbers). The inductive predicate `Int_eqm_unsigned` injects 32-bit integers to integers and the tactic `solve_Int_eqm_unsigned` generates that injection result (it also uses Coq `rewrite` tactics to improve the Coq expression).

**Computing root type**

To partially generate the root type, I use domain-specific small scale reflection again.

<table>
<thead>
<tr>
<th>Inductive compute_root_type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall (t_from_e: type) (lr: LLRR) (t_root: type), Prop :=</td>
</tr>
<tr>
<td>compute_root_type_lvalue: forall t,</td>
</tr>
<tr>
<td>compute_root_type t LLLL t</td>
</tr>
<tr>
<td>compute_root_type_Tpointer_expr: forall t a1 n a2,</td>
</tr>
<tr>
<td>compute_root_type (Tpointer t a1) RRRR (Tarray t n a2)</td>
</tr>
<tr>
<td>compute_root_type_Tarray_expr: forall t n1 a1 n2 a2,</td>
</tr>
<tr>
<td>compute_root_type (Tarray t n1 a1) RRRR (Tarray t n2 a2).</td>
</tr>
</tbody>
</table>

In this Coq predicate “compute_root_type \( t_{\text{from \ e_r}} \) \( t_{\text{r}} \)”, \( t_{\text{from \ e_r}} \) is the C type of root expression \( e_r \) and \( t_{\text{r}} \) is the root type of C aggregate type. When \( e_r \) is an l-value, these two types should be equal. When \( e_r \) is a r-value of a pointer type, \( t_{\text{r}} \) will be the corresponding array—but the array’s size is still unknown.

Although this root type is only partially determined, it is enough for later SEP clause searching.
Summary of address computation

These three assumptions together with syntactic path computation result ensure that

\[ \Sigma; \Gamma; \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \vdash \lbrack \& e \rbrack = p + \delta(\overrightarrow{f}). \]

Later typechecking conditions will guarantee that \( \overrightarrow{f} \) satisfies the field-compatible criterion. Thus,

\[ \Sigma; \Gamma; \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \vdash \lbrack \& e \rbrack = p. \overrightarrow{f}. \]

12.2.4 Improving path representation via user’s hint

Usually, the load/store address “field.address \( t_r \overrightarrow{f} \ p \)” computed from syntactic path is already useful for postcondition generation. But there are some exceptions.

I illustrate with examples in a VST application—verifying the mbedTLS implementation of AES encryption [26]. The program uses a C struct as follows (simplified):

```c
struct aes_context {
    int nr; // number of rounds
    int rk[60]; // round keys
};
```

In the following examples, we assume that a pointer named \( \text{ctx} \) of type `struct aes_context*` is in scope. Here are two ways of initializing (parts of) the round key array of this context struct:

```c
for (i = 0; i < 8; i++) {
    (*ctx).rk[i] = ...
}
```

```c
int *p = (*ctx).rk;
for (i = 0; i < 8; i++) {
    p[i] = ...
}
```
The first C implementation is most straightforward. That is, both .rk and [i] are on the left-hand side of the assignment statement. Its verification may use the following loop invariant and assertions:

\[
\text{for } (i = 0; i < 8; i++) \{
    \{ \exists i. \text{PROP}(0 \leq i < 8)\text{LOCAL}(\text{temp ctx } c; \text{temp } i)\text{SEP}(c \xrightarrow{\text{aes_context}} .) \}\ \\
    \text{Given } i. \text{ Assume } 0 \leq i < 8.
    \{ \text{PROP()}\text{LOCAL}(\text{temp ctx } c; \text{temp } i)\text{SEP}(c \xrightarrow{\text{aes_context}} .) \}\ \\
    (*ctx).rk[i] = ...
\}
\]

In this case, the address computed from syntactic path is “field_address aes_context [rk; i] c” which is good enough for postcondition generation.

The second AES implementation optimizes by precomputing the .rk part of the path before the loop. Its verification may use the following loop invariant and assertions:

\[
\text{int } *p = (*ctx).rk;
\text{for } (i = 0; i < 8; i++) \{
    \{ \exists i. \text{PROP}(0 \leq i < 8)\text{LOCAL}(\text{temp ctx } c; \\
        \text{temp } p (\text{field_address aes_context [rk] } c); \text{temp } i)\text{SEP}(c \xrightarrow{\text{aes_context}} .) \}\ \\
    \text{Given } i. \text{ Assume } 0 \leq i < 8.
    \{ \text{PROP()}\text{LOCAL}(\text{temp ctx } c; \\
        \text{temp } p (\text{field_address aes_context [rk] } c); \text{temp } i)\text{SEP}(c \xrightarrow{\text{aes_context}} .) \}\ \\
    p[i] = ...
\}
\]

As we can see, only part of the access path (namely [i]) is visible in the assignment command, while another part of the access path is above the loop. In this case, the address computed from syntactic path is:
field_address (int[60]) [i] (field_address aes_context [rk] c).

We hope that it could be rewritten into: field_address aes_context [rk; i] c.

Here is another proof style of the second implementation above:

```c
int *p = (*ctx).rk;

\ Given p. Assume p = field_address aes_context [rk] c.
\ {PROP()LOCAL(temp ctx c; temp p p)SEP(c \overleftarrow{\text{aes}_\text{context}} \rightarrow -)}

for (i = 0; i < 8; i++) {
  \{∃i. PROP(0 ≤ i < 8)LOCAL(temp ctx c; temp p p; temp i i)SEP(c \overleftarrow{\text{aes}_\text{context}} \rightarrow -)}

  Given i. Assume 0 ≤ i < 8.
  \{PROP()LOCAL(temp ctx c; temp p p; temp i i)SEP(c \overleftarrow{\text{aes}_\text{context}} \rightarrow -)}

  p[i] = ...
}
```

This time, the address of p, field_address aes_context [rk] c, is remembered as p before the loop. In the loop body, the address computed from syntactic path is: field_address (int[60]) [i] p. We hope that we can use the assumption

\[ p = \text{field_address aes_context [rk] c}, \]

which is generated by Coq’s remember tactic, to improve this address expression.

The following table summarizes these three different situations. (Samuel Gruetter first summarized these three different cases. Before that, VST-Floyd only handled the first kind of situations. He also proposed and implemented the first solution to cover all these three cases. I designed this different solution demonstrated here. It needs less Coq engineering and is much easier to maintain. Samuel Gruetter’s original solution can be found in our previous publication [17].)
<table>
<thead>
<tr>
<th>Original address expressions</th>
<th>Better address expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_address aes_context [rk; i] c</td>
<td>field.address aes_context [rk; i] c</td>
</tr>
<tr>
<td>field.address aes_context [i]</td>
<td>field.address aes_context [rk; i] c</td>
</tr>
<tr>
<td>(field_address (int[60]) [rk] c)</td>
<td>field.address aes_context [rk; i] c</td>
</tr>
<tr>
<td>field.address aes_context [i] p</td>
<td>field.address aes_context [rk; i] c</td>
</tr>
<tr>
<td>(given ( p = \text{field_address (int[60]) [rk] c} ))</td>
<td>field.address aes_context [rk; i] c</td>
</tr>
</tbody>
</table>

In response to this demand of address expression improvement, I define a Coq inductive predicate `field.address.gen`:

**Inductive** field.address.gen:

\[
\begin{align*}
\text{type} & \rightarrow \text{list gfield} \rightarrow \text{val} \\
& \rightarrow \text{type} \rightarrow \text{list gfield} \rightarrow \text{val} \rightarrow \text{Prop} \\
\end{align*}
\]

| field.address.gen.nil: forall \( t_1 t_2 \rightarrow f \rightarrow p \rightarrow \text{tgtp} \), nested.field.type \( t_2 \rightarrow f \rightarrow t_1 \rightarrow \) 
field.address.gen \( (t_2, \rightarrow f, p) \rightarrow \text{tgtp} \) 
field.address.gen \( (t_1, \rightarrow f, (\text{field.address} t_2 \rightarrow f p)) \rightarrow \text{tgtp} \) |
| field.address.gen.app: forall \( t_1 t_2 \rightarrow f_1 \rightarrow f_2 \rightarrow p \rightarrow \text{tgtp} \), nested.field.type \( t_2 \rightarrow f_2 \rightarrow t_1 \rightarrow \) 
field.address.gen \( (t_2, \rightarrow f_1 + + f_2, p) \rightarrow \) 
field.address.gen \( (t_1, \rightarrow f_1, (\text{field.address} t_2 \rightarrow f_2 p)) \rightarrow \text{tgtp} \) |
| field.address.gen.assu: forall \( t \rightarrow f \rightarrow p_1 p_2 \rightarrow \text{tgtp}, p_1 = p_2 \rightarrow \) 
field.address.gen \( (t, \rightarrow f, p_2) \rightarrow \) 
field.address.gen \( (t, \rightarrow f, p_1) \rightarrow \) |
| field.address.gen.refl: forall \( \text{tgtp}, \text{field.address.gen tgp tgp} \rightarrow \) |

This is another domain-specific small scale reflection. Its semantics can be summarized by the following Coq lemma.
**Lemma** field_address_gen_fact: forall \{cs: compspecs\} \(t_1 \stackrel{f_1}{\rightarrow} p_1 \rightarrow t_2 \stackrel{f_2}{\rightarrow} p_2\),

\[
\text{field.address.gen} (t_1, \stackrel{f_1}{\rightarrow} p_1) \rightarrow \\
\text{field.address} t_1 \stackrel{f_1}{\rightarrow} p_1 = \text{field.address} t_2 \stackrel{f_2}{\rightarrow} p_2 \land \\
\text{nested.field.type} t_1 \stackrel{f_1}{\rightarrow} = \text{nested.field.type} t_2 \stackrel{f_2}{\rightarrow} \land \\
(\text{field.compatible} t_2 \stackrel{f_2}{\rightarrow} p_2 \rightarrow \text{field.compatible} t_1 \stackrel{f_1}{\rightarrow} p_1).
\]

In short, it says that if field\_address\_gen can shift \((t_1, f_1, p_1)\) to \((t_2, f_2, p_2)\) then the field addresses that they represent are the same and the field\_compatible property is (reversely) preserved.

**Solve\_field\_address\_gen** is a Coq tactic to repeatedly apply the constructors of field\_address\_gen. It is used to prove a Coq proof with form “field\_address\_gen \((t_1, f_1, p_1)\) ?”. In this process, nested field\_address like

\[
\text{field.address} (\text{int}[60]) [i] (\text{field.address aes\_context [rk] c})
\]

will be concatenated and Coq assumptions like

\[
p = \text{field.address aes\_context [rk] c}
\]

will be used to replace \(p\) with “\text{field.address aes\_context [rk] c}” in expressing root address.

**Ltac** field\_address\_assumption :=

**match** goal with

\mid H: ?a = field\_address _ _ _ \Rightarrow \text{constr.eq a b; simple eapply H}

**end.**

**Ltac** solve\_field\_address\_gen :=

solve [ repeat first

\mid simple apply field\_address\_gen.nil; [reflexivity |]

\mid simple apply field\_address\_gen.app; [reflexivity |]

\mid simple eapply field\_address\_gen.assu; [field\_address\_assumption |]

\mid simple apply field\_address\_gen.refl ]].
12.2.5 Searching in SEP clauses and generating postconditions

Now that the load/store address is generated, we are ready to search in the SEP clauses and generate the strongest postcondition.

I define a Coq inductive predicate \texttt{find\_nth\_preds} and build a corresponding Coq tactic for searching some particular kind of elements in a list.

\textbf{Inductive} \texttt{find\_nth\_preds\_rec} \{\texttt{A: Type}\} \ (\texttt{pred: A \rightarrow Prop}):\n
\hspace{1cm} \texttt{nat \rightarrow list A \rightarrow option (nat \ast A) \rightarrow Prop} := \n
\hspace{1cm} \mid \texttt{find\_nth\_preds\_rec\_cons\_head}: \forall n R0 R, \n\hspace{2cm} \texttt{pred} R0 \rightarrow \n\hspace{3cm} \texttt{find\_nth\_preds\_rec\_pred} n (R0 :: R) (\texttt{Some} (n, R0)) \n
\hspace{1cm} \mid \texttt{find\_nth\_preds\_rec\_cons\_tail}: \forall n R0 R R\_res, \n\hspace{2cm} \texttt{find\_nth\_preds\_rec\_pred} (S n) R R\_res \rightarrow \n\hspace{3cm} \texttt{find\_nth\_preds\_rec\_pred} n (R0 :: R) R\_res \n
\hspace{1cm} \mid \texttt{find\_nth\_preds\_rec\_nil}: \forall n, \n\hspace{2cm} \texttt{find\_nth\_preds\_rec\_pred} n \texttt{nil} \texttt{None}.

\textbf{Inductive} \texttt{find\_nth\_preds} \{\texttt{A: Type}\} \ (\texttt{pred: A \rightarrow Prop}):\n
\hspace{1cm} \texttt{list A \rightarrow option (nat \ast A) \rightarrow Prop} := \n
\hspace{1cm} \mid \texttt{find\_nth\_preds\_constr}: \forall R R\_res, \n\hspace{2cm} \texttt{find\_nth\_preds\_rec\_pred} 0 R R\_res \rightarrow \n\hspace{3cm} \texttt{find\_nth\_preds\_pred} R R\_res.

The following lemma describes the semantics of \texttt{find\_nth\_preds}. That is, given a Coq type \texttt{A}, a predicate \texttt{pred} over \texttt{A} and a list \texttt{R} of \texttt{A}. We try to find a natural number \texttt{n} such that the \texttt{n}-th element of \texttt{R} satisfies \texttt{pred}.

\textbf{Lemma} \texttt{find\_nth\_preds\_Some}: \forall \{\texttt{A: Type}\} \ (\texttt{pred: A \rightarrow Prop}) \texttt{R n R0}, \n\hspace{1cm} \texttt{find\_nth\_preds\_pred} \texttt{R} (\texttt{Some} (n, R0)) \rightarrow \n\hspace{1cm} \texttt{nth\_error} \texttt{R n} = \texttt{Some} R0 \land \texttt{pred} R0.
Find.nth is a parameterized tactic. It takes an argument tac which tries solve a proof goal with form “pred R_0”. For every element R_0 in R, find.nth will send it to tac to decide whether it satisfies pred. The result of calling tac should be either “solved” or “failed”. In other words, (1) tac should never leave the proof goal unsolved (2) tac may fail even if “pred R_0” is provable but not provable within “tac”’s automation. If tac proves “pred R_0”, then it means that we have found the required term. If tac fails, then find.nth will continue to try other elements in R.

Ltac find.nth.rec tac :=
  first [ simple eapply find.nth.preds.rec.constr.head; tac
           | simple eapply find.nth.preds.rec.constr.tail; find.nth.rec tac
           | simple eapply find.nth.preds.rec.nil].

Ltac find.nth tac :=
  eapply find.nth.preds.constr; find.nth.rec tac.

For a load command which loads from “field.address t' r \rightarrow f' p””, we look for a SEP clause with form “field.at t' r \rightarrow f_0 v' p” in which f_0 is a prefix of f'; in other words, we can write f as f_0 \cdot f_1. Additionally, “data.at t' v' p’” is also a satisfactory candidate because it is equivalent with “field.at t' nil v' p’”.

I use find.nth to find such SEP clause, or technically, to fill the unification variables in the following Coq proof goal:

...  

--------------------

find.nth.preds
(fun Rn ⇒ Rn = field.at t' \rightarrow f_0 v' p' ∧ f = f_0 \cdot f_1)  
\rightarrow (Some (?n, ?Rn))

In the end, “temp x v'.f_1” will be added to the LOCAL clauses in the postcondition and original LOCAL clause about x will be deleted (if exists). Here, v'.f_1 has Coq type:
We know that it is equivalent to \texttt{val} for any load command. But in Coq, these two types are equivalent but not reducible. Thus, in our load rule, we use a John-Major equality \cite{45} casting \( v' \mapsto f'_{1} \) to Coq type \texttt{val}.

For a store command which stores value \( u \) into \texttt{field-address \( t'_{r} \mapsto f'_{p} \)}, we also look for a SEP clause with form \texttt{field-at \( t'_{r} \mapsto f'_{0} \ v' \mapsto p' \)} in which \( f'_{0} = f'_{0} \mapsto f'_{1} \). We will replace this SEP clause with \texttt{field-at \( t'_{r} \mapsto f'_{0} \ (v' \mapsto f'_{1} \mapsto u) \mapsto p' \)} in the postcondition.

Readers may notice that the \texttt{find-nth-preds} assumption in \texttt{SEMAX-LOAD-CANON} and \texttt{SEMAX-STORE-CANON} are a little bit different. This is because of the transition from \texttt{data-at} to \texttt{field-at} with an empty path. For a load command, it is not a problem to always turn a \texttt{data-at} into a \texttt{field-at}. This intermediate step will not be revealed in the generated postcondition. For a store command, it is not desirable to have a \texttt{field-at} in the postcondition when the corresponding SEP clause in the precondition is a \texttt{data-at}. In the strongest postcondition generation for store rules, the variable \( Rv \) in rule \texttt{SEMAX-STORE-CANON} will be instantiated with

\[
\text{fun } v \Rightarrow \texttt{field-at } t'_{r} \mapsto f'_{0} \ v \mapsto p'
\]

when \( Rn \) is \texttt{field-at} \( t'_{r} \mapsto f'_{0} \ v' \mapsto p' \). On the other hand, \( Rv \) will be instantiated with

\[
\text{fun } v \Rightarrow \texttt{data-at } t'_{r} \ v \mapsto p'
\]

when \( Rn \) is \texttt{data-at} \( t'_{r} \ v' \mapsto p' \).

### 12.2.6 Summary

The strongest postcondition generator for load and store commands will first \texttt{eapply} \texttt{SEMAX_LOAD_CANON} or \texttt{SEMAX_STORE_CANON}. This \texttt{eapply} turns the original proof goal into a series of subgoals in which some Coq unification variables are uninstantiated. I build Coq tactics to solve these proof goal one by one—unification variables are also filled one by one in this process.
Actually, many of these subgoals are designed to compute the values of those unification variables. Only the last two subgoals are exceptions. They check some side-conditions to ensure that the load/store command will not cause a run-time error.

12.3 Function calls

The effects of C function calls are more complicated than set, load, or store commands. A function call may assign its result to a nonaddressable variable and may modify the data stored in heap. Fortunately, separation logic allows us to reason about the behaviors of calls and the canonical form allows us to present it concisely.

In section 2.3 I presented this sample C function specification:

\[
\text{swapint}(x, y) : \text{WITH } a, b, p, q.
\]

\[
\begin{align*}
\text{PRE:} & \quad \text{PROP() LOCAL(temp x p; temp y q) SEP(p } \rightleftharpoons \text{ int } a; q \rightleftharpoons \text{ int } b) \\
\text{POST:} & \quad \text{PROP() LOCAL() SEP(p } \rightleftharpoons \text{ int } b; q \rightleftharpoons \text{ int } a)
\end{align*}
\]

Now assume this specification is in \( \Delta \).

Then consider this function call: \text{swapint}(x, x+2), in a context where \( x \) is a pointer into an array of at least three consecutive integers. We would have this proof goal:

\[
\Delta \vdash \{\text{PROP() LOCAL(temp x x) SEP(x } \rightleftharpoons \text{ int } u; \ x + 4 \rightleftharpoons \text{ int } v; \ x + 8 \rightleftharpoons \text{ int } w)\}
\]

\[
\text{swapint}(x, x + 2)
\]

\[
\{?\text{Post, [\perp]}\}
\]
The strongest postcondition is,

\[ \text{PROP}(\text{LOCAL}(\text{temp } \times x) \text{SEP}(x + 4 \mapsto v; \ x \mapsto w; \ x + 8 \mapsto u)) \]

How can this be generated? First, the user instantiates the parameters \((a, b, p, q)\) with the values \((u, w, x, x + 8)\). The instantiated specification is:

\[
\text{swapint}(x, y) : \text{WITH } a, b, p, q.
\]

\[
\text{PRE: } \text{PROP}(\text{LOCAL}(\text{temp } x; \text{temp } y (x + 8)) \text{SEP}(x \mapsto u; \ x + 8 \mapsto w))
\]

\[
\text{POST: } \text{PROP}(\text{LOCAL}() \text{SEP}(x \mapsto w; \ x + 8 \mapsto u))
\]

Second, the \text{LOCAL} part of the precondition in this specification is verified automatically (and computationally) by C expression evaluation, i.e.

\[
\begin{align*}
\text{msubst_eval_expr } [\text{temp } x ] (x) &= x \\
\text{msubst_eval_expr } [\text{temp } x ] (x + 2) &= x + 8
\end{align*}
\]

(Since \(x+2\) in C is a pointer-integer add, the semantics of C gives the address \(x + 8\).)

Third, for the \text{SEP} part, we pick out the precondition of the instantiated specification in the precondition of the proof goal, replace it with the postcondition of the instantiated specification and use the replacement result as the generated postcondition in the proof goal. This “picking out” is a form of “frame inference,” and is accomplished by a cancellation tactic.

Fourth, this function call has no return value, so the \text{LOCAL} part of the generated postcondition is the same as the precondition.

We mostly automate this process in VST-Floyd, but require our users to manually instantiate the parameters in the specification. The soundness of this process is ensured by the following Hoare rule, \text{SEMAX-CALL-00}. We prove it in Coq as a derived rule of Verifiable C.
Lemma SEMAX\_CALL\_00:

\[
\begin{align*}
\forall \Delta \overset{\varphi}{\rightarrow} R \overset{f}{\rightarrow} e' \ a \ \text{Pre Post} \overset{P_{\text{pre}}}{\rightarrow} R_{\text{pre}} \overset{P_{\text{post}}}{\rightarrow} R_{\text{post}} f, \\
(f(\vec{x}): \text{WITH}_{a: A. \{\text{Pre}(a)\} \{\text{Post}(a)\}}) \in \Delta \rightarrow \\
\text{Pre } a = \text{PROP}(\vec{P}_{\text{pre}}) \text{ LOCAL}(\text{temp_list } \vec{x} \ \vec{v}) \text{ SEP}(\vec{R}_{\text{pre}}) \rightarrow \\
\text{Post } a = \exists b: B, \text{PROP}(\vec{P}_{\text{post}}(b)) \text{ LOCAL}() \text{ SEP}(\vec{R}_{\text{post}}(b)) \rightarrow \\
\vec{P}_{\text{pre}} \rightarrow \\
\text{PROP()} \text{ LOCAL}(\vec{Q}) \text{ SEP}(\vec{R}) \vdash \text{tc_expr_list } \vec{e} \rightarrow \\
\text{msubst_eval_expr_list } \vec{e} \vec{Q} = \text{Some } \vec{v} \rightarrow \\
\text{Permutation } \vec{R} (\vec{P}_{\text{pre}} \text{ ++ } \vec{F}) \rightarrow \\
\text{semax } \Delta (\text{PROP()} \text{ LOCAL}(\vec{Q}) \text{ SEP}(\vec{R})) \\
(f(\vec{e})) \\
(\exists b: B, \text{PROP}(\vec{P}_{\text{post}}(b)) \text{ LOCAL}(\vec{Q}) \text{ SEP}(\vec{R}_{\text{post}}(b) \text{ ++ } \vec{F}), \bot)
\end{align*}
\]

A C function may or may not return a value. If it does return a value, the call site may or may not assign that value to a variable. Therefore, we have three cases:

11 The function returns a value, and the call site assigns it to a variable.

01 The function returns a value, but the call site throws it away.

00 The function does not return a value, and (therefore in a well typed C program) the call site does not expect a value.

The example above (with \texttt{semex\_call\_00}) is of the third kind. The derived lemmas for the other two kinds are similar, and have names ending with 11 and 01.

In the \texttt{swapint} example above, the PROP part of the precondition in the specification is empty. If it were not empty, our postcondition generator would check these pure facts.

The postconditions in specifications can be existentially quantified (which does not happen in our example above). Our postcondition generator and the derived Hoare rules do cover those cases. Generally speaking, the generated postcondition is
an existentially quantified canonical assertion. When the postcondition in a specification is not quantified, we treat it as quantified over unit type. When the generated postcondition is quantified over unit type, our generator removes the quantifier and presents a quantifier-free version.

Comparing to the cases for set, load or store, the generator for function calls imposes more restrictions on the user. Besides the fact that users need to manually provide values to instantiate the specification, users also need to ensure an exact match between specification SEP clauses and a subset of the SEP clauses in the proof goal. Specifically, when handling load or store commands, we do detect \( p \xrightarrow{\text{fst}} a \) inside \( p \xrightarrow{\text{IntPair}} a, b \). But we choose not to support a similar feature when handling function calls because the situation here is much more complicated. For example, the correspondence between the specification and the precondition in the proof goal may be many-to-one. It is very hard to detect

\[
\text{SEP}(p.\text{fst} \xrightarrow{\text{int}} a; p.\text{snd} \xrightarrow{\text{int}} b)
\]

inside \( p \xrightarrow{\text{IntPair}} a, b \). We require users to apply related transformations in preconditions first. VST-Floyd provides proof rules for these transformations (see section 10.2.1).
Chapter 13

Forward verification

The most important feature of VST-Floyd is its forward proof style. VST-Floyd provides a set of tactics to perform forward verification. From one point of view, these tactics help users build proof trees of Hoare triples and automate the routine work. From another point of view, building proofs of Hoare triples using forward tactics is like demonstrating a decorated program from the top down. In the beginning of this chapter, I will start with two typical tactics in VST-Floyd for forward verification—forward (section 13.1) and forward_call (section 13.2). Each call to them is like demonstrating one more C assignment command and one more assertion in a decorated program.

13.1 Forward through set, load and store commands

VST-Floyd offers a tactic forward to perform forward verification on set, load and store commands.

The tactic forward analyzes the first command in a triple and proceeds by forward reasoning to shrink the proof goal. Fig. 13.1 and 13.2 shows one example of applying
Figure 13.1: Before executing forward

\[
\Delta \vdash \begin{cases}
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times p; \text{temp} y a) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto 0 \end{array} \right)
\end{cases}
\begin{align*}
z &= \ast x \\
y &= y + z \\
x &= x + 1 \\
\ast x &= y
\end{align*}
\]

\[
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times (p + 4)) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto a + b \end{array} \right)
\]

Figure 13.2: After executing forward. Floyd has made a local definition \textit{MORE.COMMANDS} so that the proof goal below the line is not cluttered with the entire remainder of the block.

\[
\Delta \vdash \begin{cases}
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times p; \text{temp} y a) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto 0 \end{array} \right)
\end{cases}
\begin{align*}
z &= \ast x \\
y &= y + z \\
x &= x + 1 \\
\ast x &= y
\end{align*}
\]

\[
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times (p + 4)) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto a + b \end{array} \right),
\]

\[\vec{\perp}\]

\textit{MORE.COMMANDS} :=
\begin{align*}
x &= x + 1; \\
\ast x &= y
\end{align*}

\[
\Delta \vdash \begin{cases}
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times p; \text{temp} y a; \text{temp} z b) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto 0 \end{array} \right)
\end{cases}
\begin{align*}
y &= y + z; \\
\textit{MORE.COMMANDS}
\end{align*}
\]

\[
\text{PROP()} \\
\text{LOCAL}(\text{temp} \times (p + 4)) \\
\text{SEP} \left( \begin{array}{c}
p \mapsto b \\
p + 4 \mapsto a + b \end{array} \right)
\]

144
forward. The Coq proof goal on the right side of Fig. 13.1 is the proof goal before executing forward. The one on the right side of Fig. 13.2 is the proof goal after executing forward. The decorated programs on the left correspond to the verification process on the right; the shaded lines correspond to the Coq proof goals.

Generally speaking, if the first command in the proof goal is an assignment (set, load or store command) \( c \) and the precondition is \( P \), forward will eliminate \( c \) and replace the precondition with the strongest postcondition of \( P \) and \( c \). Fig. 13.3 and Fig. 13.4 demonstrate this process. In Fig. 13.3, there is more than one command in the original proof goal. Applying forward reduces the proof goal to a new triple in which the new precondition \( Q \) is the strongest postcondition of \( P \) and \( c \). In Fig. 13.4, \( c \) is the only command in the original proof goal. Applying forward reduces the proof goal to a separation logic entailment, \( Q \vdash R \).

\[
\begin{array}{c}
\{P\} \\
c; \\
c_2 \\
\{R, [\vec{S}]\}
\end{array}
\quad \Delta \vdash \\
\begin{array}{c}
\{P\} \\
c; c_2 \\
\{R, [\vec{S}]\}
\end{array}
\]

\[
\begin{array}{c}
\{P\} \\
c; \\
\{Q\} \\
c_2 \\
\{R, [\vec{S}]\}
\end{array}
\quad \Delta \vdash \\
\begin{array}{c}
\{Q\} \\
c_2 \\
\{R, [\vec{S}]\}
\end{array}
\]

Figure 13.3: More than one command

13.1.1 Implementation.

The implementation of forward contains the following steps:

First, forward analyzes the C command in the proof goal. If it has form \((c_1; c_2)\) and \( c_1 \) is an assignment, then forward will do

\[\text{eapply semax_seq'}\]
If it is a single assignment command \( c \), then \textbf{forward} will do

\texttt{eapply semax\_post'}

Here, \texttt{SEMAX\_SEQ'} and \texttt{SEMAX\_POST'} are derived Hoare rules. Coq’s \texttt{eapply} tactic may create unification variables, in this case the intermediate assertion \( Q \).

**Theorem SEMAX\_SEQ':** \( \text{forall} \; \Delta \; P \; Q \; R \; \vec{S} \; c_1 \; c_2, \)

\[\text{semax} \; \Delta \; P \; c_1 \; (Q, [\vec{S}]) \rightarrow\]

\[\text{semax} \; \Delta \; Q \; c_2 \; (R, [\vec{S}]) \rightarrow\]

\[\text{semax} \; \Delta \; P \; (c_1; c_2) \; (R, [\vec{S}]).\]

**Theorem SEMAX\_POST':** \( \text{forall} \; \Delta \; P \; Q \; R \; \vec{S} \; c, \)

\[\text{semax} \; \Delta \; P \; c \; (Q, [\vec{L}]) \rightarrow\]

\[Q \vdash R \rightarrow\]

\[\text{semax} \; \Delta \; P \; c \; (R, [\vec{S}]).\]

Either way, two proof goals will be generated. The first one has form:

\[\Sigma; \Gamma; \Delta \vdash \{P\} \; c \; \{?Q, [\vec{L}]\}\]

and the second one is either \( \Sigma; \Gamma; \Delta \vdash \{?Q\} \; c_2 \; \{R, [\vec{S}]\} \) or \( \Sigma; \Gamma; ?Q \vdash R \), respectively.

The unification variable \(?Q\) is to be filled in later.
In chapter 12, I described our Ltac programs which can generate the strongest postcondition for an assignment and solve the first proof goal. *Forward* calls these Ltac programs and instantiates $?Q$. As a result, the second proof goal, fully instantiated, is presented to users.

*Forward* requires the original precondition to be in canonical form and ensures that the new precondition is also canonical.

*Forward* is mostly automatic: most premises and side conditions of rules such as SEMAX-STORE-CANON are proved automatically. But sometimes users must prove a side condition (e.g., that some expression evaluation does not cause run time error) or offer some hint (like which data_at or field_at is loaded from or stored to, see section 12.2.4).

13.1.2 Reassociating sequences

The first step in *forward* is to find the first command in a sequence of commands. Clight formalizes sequential composition as a (deeply embedded) binary syntactic operator. If the proof goal is \( \{P\} c_1; (c_2; c_3) \{Q\} \) where \( c_1 \) is an assignment statement, *forward* can use the SEMAX_SEQ’ rule to produce the proof goals \( \{P\} c_1 \{?U\} \) and \( \{?U\} c_2; c_3 \{Q\} \), where \( ?U \) is a unification variable to be filled in.

But if the goal is \( \{P\} (c_1; c_2); c_3 \{Q\} \), then SEMAX_SEQ’ cannot be applied directly. In such situation, *forward* first reorganizes the sequential composition using this rule:

**Theorem** SEQ ASSOC: forall \( \Delta P Q \bar{R} c_1 c_2 c_3 \),

\[
\text{semax} \Delta P (c_1; (c_2; c_3)) (Q, [\bar{R}]) \leftrightarrow \\
\text{semax} \Delta P ((c_1; c_2); c_3) (Q, [\bar{R}]).
\]
13.2 Forward through function calls

Forward_call verifies a function call. Its interface and implementation are very similar to forward, except that it takes an argument: the instantiation of the universally quantified parameter of the function specification. We have automated the instantiation of specification parameters.

The following is a specification for swapint, which swaps the numbers stored in two different addresses. I have used it multiple times in this thesis.

\[
\text{swapint}(x, y) : \\
\text{WITH } a, b, p, q. \\
\text{PRE: } \text{PROP} \text{ LOCAL}(\text{temp } x \ p; \text{temp } y \ q) \text{ SEP}(p \mapsto a; q \mapsto b) \\
\text{POST: } \text{PROP} \text{ LOCAL} \text{ SEP}(p \mapsto b; q \mapsto a)
\]

Figures 13.5 and 13.6 illustrate the effect of running “forward_call (2a, a^2 + 1, p, p + 4)”.

VST-Floyd requires the postconditions in function specifications to be in existentially quantified canonical form. Thus, the new precondition may have such existentials. Users can use Intros (see section 13.6) at the end of forward_call so that the triple left as the proof goal has a canonical precondition.
13.3 Forward proof through if commands

In this section and the next one, I will introduce the forward verification tactics for C structured commands—`if` commands and loops. VST-Floyd’s users can use these two tactics to reduce the current Hoare triple into several smaller ones. From another point of view, these tactics help users write decorated C programs in order.

**Forward_if** is the tactic to perform forward verification when the first C command in the proof goal is an `if` command. In general, after forward proof through two branches of an `if`, one needs to merge the postconditions together. One could say the postcondition is just a disjunction in separation logic, but that just reduces to the problem of eliminating disjunctions and returning to canonical form. So we require
the user to provide the joined postcondition as an argument to forward_if; except when the if command is the only command in the proof goal (e.g., the last command in a block), in which case the postcondition is already provided.

Consider the following sample program and specification in which p is an integer pointer and s, x and t are C integers.

\[ \Sigma; \Gamma; \Delta \vdash \{ PROP() \ LOCAL(\text{temp} \ x; \ \text{temp} \ p \ p; \ \text{temp} \ s \ \sigma) \ SEP(p \xrightarrow{\text{int}} 0) \} \]

\[
\begin{align*}
&\text{if } (x >= 0) \text{ then } *p = x; \ \text{else } *p = -x; \\
&t = *p; \ s = t + s; \\
&\{ PROP() \ LOCAL(p \ p; \ \text{temp} \ s \ (\sigma + |x|) \ SEP(p \xrightarrow{\text{int}} |x|), \ [\vec{\bot}] \} 
\end{align*}
\]

The if command in this example stores the absolute value of \([x]\) into the address \([y]\).

We can apply forward_if (PROP() LOCAL(temp x; temp p p; temp s \ \sigma) SEP(p \xrightarrow{\text{int}} |x|)).

Then three subgoals are left for us to prove (see Fig. 13.7): one for the if-then branch, one for the if-else branch and one for the C commands afterwards.
\(\Sigma; \Gamma; x \geq 0; \Delta \vdash \{ \text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ p; \text{temp} \ s \ \sigma) \text{ SEP}(p \rightarrow \int 0)\}\)

\*p = x;
\{ \text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ p; \text{temp} \ s \ \sigma) \text{ SEP}(p \rightarrow \int |x|), [\vec{I}]\}\n
\(\Sigma; \Gamma; x \not\geq 0; \Delta \vdash \{ \text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ p; \text{temp} \ s \ \sigma) \text{ SEP}(p \rightarrow \int 0)\}\)

\*p = -x;
\{ \text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ p; \text{temp} \ s \ \sigma) \text{ SEP}(p \rightarrow \int |x|), [\vec{I}]\}\n
\(\Sigma; \Gamma; \Delta \vdash \{ \text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ p; \text{temp} \ s \ \sigma) \text{ SEP}(p \rightarrow \int |x|)\}\)

t = \*p; s = t + s;
\{ \text{PROP()} \text{ LOCAL}(\text{temp} \ p \ p; \text{temp} \ s \ (\sigma + |x|) \text{ SEP}(p \rightarrow \int |x|), [\vec{I}] \}\}

Figure 13.7: Subgoals after \textit{forward-if}

Fig. 13.8 and 13.9 is a sketch of the effect of “\textit{forward-if} Q” in general. One important detail in this tactic interface is how we handle the predicate that “the denotation of \(b\) is true/false”. Traditionally, \([b] = \text{true}\) will be a conjunct of the precondition in the if-then branch. In VST-Floyd, it does not show up in the precondition but appears above the line as an assumption of the whole if-then triple. In our example above, the precondition is

\text{PROP()} \text{ LOCAL}(\text{temp} \ x; \text{temp} \ s \ s) \text{ SEP}(s \rightarrow \int \sigma) ≤ 0\) (a C expression of type \text{bool}). Thus “\(x \geq 0\)” (a Coq proposition) will an assumption of the if-then triple. Similarly, “\(x \not\geq 0\)” (another Coq proposition) will be an assumption of the if-else triple. Treating this testing result as a Coq assumption about values, instead of a Hoare logic precondition about expressions, is very convenient in verifying real C programs. For example, if the Coq assumption
is an equality, then Coq’s tactics like subst and rewrite can use such assumptions directly.
Implementation. In “\texttt{forward\_if }Q\texttt{ “}, we first apply \texttt{SEMAX-SEQ} to split the proof goal into two. The first one will be handled by \texttt{forward\_if’} and the second one, a triple for the rest of program is directly left to the users (as the third proof goal).

The implementation of \texttt{forward\_if’} is based on the following auxiliary Hoare rule.

\textbf{Lemma} \texttt{SEMAX\_IF’}: \texttt{forall} \(\Delta \vdash Q \rightarrow R\) \(v b c_1 c_2\) \(\text{Post}\),

\begin{align*}
\text{PROP () LOCAL } Q & \text{ SEP } R \vdash \text{tc\_expr } \Delta \ b \rightarrow \\
\text{msubst\_eval\_expr } b \ Q & = \text{Some } v \\
\text{(typed\_true } v & \rightarrow \text{semax } \Delta \ (\text{PROP () LOCAL } Q \text{ SEP } R) \ c_1 \ \text{Post}) \rightarrow \\
\text{(typed\_false } v & \rightarrow \text{semax } \Delta \ (\text{PROP () LOCAL } Q \text{ SEP } R) \ c_2 \ \text{Post}) \rightarrow \\
\text{semax } \Delta \ (\text{PROP () LOCAL } Q \text{ SEP } R) \ (\text{if } b \text{ then } c_1 \text{ else } c_2) \ \text{Post}.
\end{align*}

We do \texttt{ eapply semax\_if’} first in \texttt{forward\_if’}. Four subgoals are generated. The first one is typechecking; it will usually be solved automatically. The second is \texttt{C} expression evaluation, which solves by computation and \textit{v} will be instantiated using \texttt{solve\_msubst\_eval\_expr} (see section [11.4]).

The last two proof goals are the then-clause Hoare triple and the else-clause Hoare triple. Simply doing \texttt{intro} can pull the proposition “\texttt{typed\_true }\textit{v}” or “\texttt{typed\_false }\textit{v}” into Coq assumption lists. Afterwards, these two Hoare triples are presented to users.

When the \texttt{if} command is the last command in the triple in the proof goal, \texttt{forward\_if} (without arguments) should be used. Its implementation is directly calling \texttt{forward\_if’}.

\footnote{Since \texttt{bool} and \texttt{Prop} are two different type in Coq and using the latter one is more convenient, \texttt{forward\_if’} actually does more than \texttt{intro} to this two subgoals. Usually, the Coq expression of \textit{v} (which is generated by \texttt{solve\_msubst\_eval\_expr}) will be boolean related, e.g. “\texttt{Val\_of\_bool (Int\_eqb } x \ y\texttt{ )}”—here, \texttt{Int\_eqb} returns a boolean representing whether \textit{x} and \textit{y} are equal. In this case, “\texttt{typed\_true (Val\_of\_bool (Int\_eqb } x \ y\texttt{ )}” will be transformed to “\textit{x=y}” before \texttt{intro}.}
13.4 Forward proof through loops

CompCert Clight unifies different loops in C into the form loop($c_i$) $c$. In this general loop command, $c$ is the loop body and $c_i$ is the increment command. Specifically, “while ($b$) $c$” is defined as

$$\text{loop}(); \{ \text{if ($b$) /*skip*/; else break; } c \}$$
and “for ($c_0$; $b$; $c_i$) $c$” is defined as

$$c_0; \text{loop}({c_i}) \{ \text{if ($b$) /*skip*/; else break; } c \}.$$

As explained earlier (section 2.6.3), Verifiable C offers a primary Hoare rule for general loops—semax-loop. Here is its Coq formalization.

**Lemma SEMAX_LOOP**: for all $\Delta$ $c$ $I$ $I_{con}$ $Q$ $Q_{brk}$ $Q_{con}$ $Q_{ret}$,

$$\text{semax} \Delta I c (I_{con}, [Q, I_{con}, Q_{ret}]) \to$$

$$\text{semax} \Delta I_{con} c_i (I, [Q, \bot, Q_{ret}]) \to$$

$$\text{semax} \Delta I \text{loop} (c_i) c (Q, [Q_{brk}, Q_{con}, Q_{ret}]).$$

When the proof goal has the following form (i.e. the first command in the triple is a loop):

$$\Sigma; \Gamma; \Delta \vdash \{P\} \text{loop}({c_i}) c; \text{MORE_COMMANDS} \{R, [S_{brk}, S_{con}, S_{ret}]\}$$

Users can use VST-Floyd’s tactic: forward_loop $I$ continue: $I_{con}$ break: $Q$. This tactic simply applies the sequence rule, consequence rule and semax-loop. The proof goals left for users are:

$$\Sigma; \Gamma; P \vdash I$$

$$\Sigma; \Gamma; \Delta \vdash \{I\} c \{I_{con}, [Q, I_{con}, S_{ret}]\}$$

$$\Sigma; \Gamma; \Delta \vdash \{I_{con}\} c_i \{I, [Q, \bot, S_{ret}]\}$$

$$\Sigma; \Gamma; \Delta \vdash \{Q\} \text{MORE_COMMANDS} \{R, [S_{brk}, S_{con}, S_{ret}]\}$$
When the proof goal has the following form (i.e. the only command in the triple is a loop):

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \ \text{loop}(c_i) \ c \ \{ R, [S_{\text{brk}}, S_{\text{con}}, S_{\text{ret}}] \}
\]

Users can use \textbf{forward\_loop} without the \textbf{break} clause: \textbf{forward\_loop} \textit{I} \ \textbf{continue}: \textit{I}_{\text{con}}. In this case, the proof goals left for users are:

\[
\begin{align*}
\Sigma; \Gamma; P \vdash \textit{I} \\
\Sigma; \Gamma; \Delta \vdash \{ \textit{I} \} \ c \ \{ \textit{I}_{\text{con}}, [R, \textit{I}_{\text{con}}, S_{\text{ret}}] \} \\
\Sigma; \Gamma; \Delta \vdash \{ \textit{I}_{\text{con}} \} \ c_i \ \{ \textit{I}, [R, \bot, S_{\text{ret}}] \}
\end{align*}
\]

### 13.5 Weakening preconditions

In Hoare logic proofs, \textbf{HOARE-CON} is a useful rule to connect parts together:

\[
\text{If } \Sigma; \Gamma; \Delta \vdash \{ P \} \ c \ \{ Q, [\vec{R}] \},
\]

\[
\text{HOARE-CON: } \Sigma; \Gamma; P' \vdash P, \ \Sigma; \Gamma; Q \vdash Q' \text{ and for any } i, \Sigma; \Gamma; R_i \vdash R_i' \text{ then } \Sigma; \Gamma; \Delta \vdash \{ P' \} \ c \ \{ Q', [\vec{R}'] \}.
\]

In a decorated program, we write two consecutive assertions (the second one should be derivable from the first one) to represent an application of this consequence rule. Correspondingly, users need to weaken the precondition of a Hoare triple or the left side of an entailment. Figure 13.10 and 13.11 illustrate this correspondence between writing decorated programs and building forward verification in VST-Floyd.
\[ \Delta \vdash \{ P \} \text{COMMANDS} \{ R, [\vec{S}] \} \]

\[ \Delta \vdash \{ Q \} \text{COMMANDS} \{ R, [\vec{S}] \} \]

Figure 13.10: Weaken a Hoare Triple’s Precondition

\[ \Delta \vdash \{ P \} \text{COMMANDS} \{ R, [\vec{S}] \} \]

\[ P \vdash Q \text{ (* Side Condition, If Necessary *)} \]

\[ \Delta \vdash \{ Q \} \text{COMMANDS} \{ R, [\vec{S}] \} \]

Figure 13.11: Weaken an Entailment’s Left Side

13.5.1 Leveraging Coq’s built-in tactics

Coq provides a lot of tactics to replace one term with another provably equal one: rewrite, replace, change, subst, simpl, unfold, fold etc.\[8\]. For example, in the wandQ-frame proof for linked list append in Chapter 6.2, the transition from line 33 to 34 weakens the precondition of a Hoare triple. It transforms

\[ [x] = x \land \text{listrep}(x, l'_1 \cdot \text{cons}(a, l_2)) \]
which in canonical form is

\[
\text{PROP()} \ \text{LOCAL}(\text{temp \ } x) \ \text{SEP}(\text{listrep}(x, l'_1 \cdot \text{cons}(a, l_2)))
\]

to

\[
[x] = x \ \land \ \text{listrep}(x, l_1 \cdot l_2)
\]

which in canonical form is

\[
\text{PROP()} \ \text{LOCAL}(\text{temp \ } x) \ \text{SEP}(\text{listrep}(x, l_1 \cdot l_2))
\]

given the following assumptions:

\[
H_1 : l_1 = l'_1 \cdot \text{cons}(a, l''_1) \quad H_2 : l''_1 = \text{nil}.
\]

This transformation can be done by:

\[
\text{subst } l''_1; \ \text{simpl in } H_1; \ \text{rewrite cons.app, appassoc, } \leftarrow \ H_1.
\]

Here, \text{subst } l''_1; \ \text{simpl in } H_1 \text{ turns the assumption } H_1 \text{ into } l_1 = l'_1 \cdot [a]. \ And \ \text{cons.app} \text{ and } \text{appassoc} \text{ are two theorems about list concatenation—their Coq formalization is:}

\textbf{Theorem appassoc}: \forall (A : \text{Type}) \ (l \ m \ n : \text{list } A),
\[
l \cdot (m \cdot n) = (l \cdot m) \cdot n.
\]

\textbf{Theorem cons.app}: \forall (A : \text{Type}) \ (x : A) \ (y : \text{list } A),
\[
\text{cons}(x, y) = [x] \cdot y.
\]

13.5.2 \textbf{Gather SEP and Replace SEP}

\textit{VST-Floyd} provides \texttt{gather.SEP} and \texttt{replace.SEP} to manipulate \texttt{SEP} clauses in preconditions or in the left sides of entailments. This is in order to replace the precondition (or the left side of an entailment) with a weaker one (not a provably equivalent one)—
Coq’s built-in tactics cannot achieve this in most cases. For example, suppose the proof goal is a triple (or an entailment, respectively) whose precondition (or left side, respectively) is

$$\text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_0, R_1, R_2, R_3, R_4, R_5, R_6))$$

$\text{gather}_\text{SEP} \ i \ j \ k$ will bring the $i$th, $j$th, and $k$th items to the front of the $\text{SEP}$ list and conjoin them into a single element.

- $\text{gather}_\text{SEP} \ 5$ results in $\text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_5, R_0, R_1, R_2, R_3, R_4, R_6))$.
- $\text{gather}_\text{SEP} \ 1 \ 3$ results in $\text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_1 * R_3, R_0, R_2, R_4, R_5, R_6))$.
- $\text{gather}_\text{SEP} \ 3 \ 0$ results in $\text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_3 * R_0, R_1, R_2, R_4, R_5, R_6))$.

$\text{replace}_\text{SEP} \ i \ R^*$ will replace item $\#i$ with predicate $R^*$.

- $\text{replace}_\text{SEP} \ 5 \ R^*$ results in $\text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_0, R_1, R_2, R_3, R_4, R^*, R_6))$.

and a proof goal:

$$\Sigma; \Gamma; \ \text{PROP}(\text{LOCAL} \overrightarrow{Q} \text{SEP}(R_5)) \vdash R^*$$

In the wandQ-frame proof for linked list append in Chapter 6.2, the transition from line 20 to 21 weakens the left side of an entailment. It transforms

$$[x] = x \land [y] = y \land [t] = u \land [u] = p \land$$

$$t \mapsto a * t + 4 \mapsto u * u \mapsto b * u + 4 \mapsto p *$$

listrep$(p, l''1) * listrep$(y, l_2) * lseg$(x, t, l'_1)$

To

$$[x] = x \land [y] = y \land [t] = u \land [u] = p \land$$

lseg$(t, u, [a]) * u \mapsto b * u + 4 \mapsto p *$

listrep$(p, l''1) * listrep$(y, l_2) * lseg$(x, t, l'_1)$.
Or equivalently, it transforms:

\[
\text{PROP ()} \\quad \text{LOCAL (temp x; temp y; temp t; temp u p)} \\quad \text{SEP (t} \mapsto \mapsto a; t + 4 \mapsto u; u \mapsto b; u + 4 \mapsto p; \\quad \text{listrep(p, l''_1); listrep(y, l_2); lseg(x, t, l'_1))}
\]
to

\[
\text{PROP ()} \\quad \text{LOCAL (temp x; temp y; temp t; temp u p)} \\quad \text{SEP (lseg(t, u, [a])); u \mapsto b; u + 4 \mapsto p; \\quad \text{listrep(p, l''_1); listrep(y, l_2); lseg(x, t, l'_1))}
\]

Users can achieve this tranformation by:

\[
\text{gather.SEP 0 1; replace.SEP 0 (lseg(t, u, [a])).}
\]

The following proof goal will be generated as a side condition:

\[
\Sigma; \Gamma; \quad \text{PROP()LOCAL(…)} \quad \text{SEP(t} \mapsto a \ast t + 4 \mapsto u) \vdash \text{lseg(t, u, [a])}.
\]

It can be proved by rule \((6.1b)\): for any \(p, q\) and \(a\),

\[
\Sigma; \Gamma; \quad p \mapsto a \ast p + 4 \mapsto q \vdash \text{lseg(p, q, [a])}.
\]

### 13.5.3 Flattening the \texttt{SEP} clauses

In canonical preconditions (or left side assertions of entailments), there may be \texttt{SEP} clauses which themselves are separating conjunctions. In contrast to gathering several \texttt{SEP} clauses together, it will be helpful sometimes to split those \texttt{SEP} clauses into separate conjuncts.
VST-Floyd provides a tactic `Intros` to achieve such transformation. It is sound because of the commutativity and associativity of separating conjunction. Fig. 13.12 shows an example.

The readers may wonder about the tactic’s name: `Intros`. The reason for this nomenclature is: to flatten the `SEP` clauses is only `Intros`’s minor effect. I will further introduce this tactic in the next section (section 13.6).

Figure 13.12: Example: Intros

13.6 Extracting existentials and propositions

Most forward verification tactics introduced above do not change the proof context ($\Sigma$) and logical assumptions ($\Gamma$)—which corresponds to the Coq assumptions in the Coq proof goals. The only exception is `forward_if`. It will add “the loop condition is true” to the assumption set of the if-then branch’s proof goal and add “the loop condition is false” to the assumption set of the if-else branch’s proof goal.
In this section and the next, I will introduce the forward verification tactics which manipulate the context.

**Intros** is the basic tactic to extract PROP clauses in preconditions, to pull out existentially quantified variables and put them into Coq assumptions. Also, it improves the arrangement of canonical preconditions (see section 13.5.3).

This tactic is necessary because most forward tactics assume that the precondition in the proof goal is in canonical form, but some forward tactics may generate an existentially quantified canonical precondition. For example, `forward-call` may generate existential quantifiers in preconditions. Also, typical loop invariants are in existentially quantified canonical form. Thus, verification of loop bodies usually starts with quantified preconditions.

### 13.6.1 Extracting PROP clauses

The first effect of **Intros** is extracting PROP clauses. Fig. 13.13 is such an example. The right side of it shows the Coq proof goals before and after applying **Intros**: two propositions $a \geq 0$ and $b \geq 0$ are dragged above the line and become assumptions of the whole triple. The corresponding decorated program says intuitively: “from this point to the end of current block, let’s assume $a \geq 0$ and $b \geq 0$”. Such transformations are sound due to **HOARE-PURE** (presented in section 2.6.3). The following is this proof rule and the Coq formalization of its canonical form version.

**HOARE-PURE:**

If $\Sigma; \Gamma; P^{\text{Pure}}; \Delta \vdash \{P\} c \{Q, [\vec{R}]\}$

then $\Sigma; \Gamma; \Delta \vdash \{P^{\text{Pure}} \land P\} c \{Q, [\vec{R}]\}$.

**Lemma** semax_extract_PROP: forall $\Delta$ $P^{P^{\text{Pure}}}$ $\vec{P}$ $\vec{Q}$ $\vec{R}$ $c$ Post,

$(P^{P^{\text{Pure}}} \rightarrow \text{semax} \Delta (\text{PROP } \vec{P} \text{ LOCAL } \vec{Q} \text{ SEP } \vec{R}) c \text{ Post}) \rightarrow$

$\text{semax} \Delta (\text{PROP } (P^{P^{\text{Pure}} :: \vec{P}}) \text{ LOCAL } \vec{Q} \text{ SEP } \vec{R}) c \text{ Post}$.

It is a very practical proof strategy in VST-Floyd to pull propositions from the precondition into Coq assumptions. It enables users to apply domain-specific math-
Figure 13.13: Example: Intros

ematics in a more flexible environment, i.e. Coq’s proof mode. For example, \(a \geq 0\), \(b \geq 0\) and \(a + b = 0\) together tell us \(a = b = 0\). This can be proved directly by Coq’s default solver for linear programming \textit{omega}. Moreover, if assumptions like \(a = 0\) are above the line, Coq tactics like \textit{subst} can be applied to further simplify the proof goal.
13.6.2 Extracting existentials

**Intros** also extracts existentially quantified variables. If the proof goal is a Hoare triple, “Intros \( x \ y \)” moves existential quantifiers in the precondition above the line, introducing Coq variables \( x, y \).

One or more arguments after **Intros** indicate the name of the new Coq variables. Fig. [13.14](#) is a tiny example. **Intros** also finds existential quantifiers inside a **SEP** clause (see Fig. [13.15](#)).

---

**Figure 13.14: Example: Intros \( a \ b \)**
Intros works (soundly) by applying HOARE-EXISTS (presented in section 2.6.3):

If $\Sigma; x; \Gamma \vdash \{P\} \ c \ \{Q, \vec{R}\}$ and

HOARE-EXISTS: $x$ does not freely occur in $\Gamma, Q$ or $\vec{R}$

then $\Sigma; \Gamma; \Delta \vdash \{\exists x. P\} \ c \ \{Q, \vec{R}\}$.

Moving existential quantifiers out of one SEP conjunct is sound due to the commutativity between existential quantifier and separating conjunction: $(\exists a. P(a)) \ast Q \vdash \exists a. P(a) \ast Q$. We prove the following derived rule for canonical assertions. More specifically, our Ltac program always moves an existentially quantified SEP conjunct.
to the beginning, then applies this rule.

\[ \Sigma; \Gamma; \text{PROP} \rightarrow P \; \text{LOCAL} \rightarrow Q \; \text{SEP}(\exists x. R_0(x); \overrightarrow{R}) \vdash \exists x. \text{PROP} \rightarrow P \; \text{LOCAL} \rightarrow Q \; \text{SEP}(R_0(x); \overrightarrow{R}) \]

13.6.3 Common scenario: **Intros** after unfolding user-defined predicates

It may seem strange that extracting existentials and flattening \text{SEP} clauses are built in one single tactic Intros. But the following sample program explains this design choice. This is a proof segment in link-list-append’s verification. It corresponds to line 15 to 20 in section 6.2’s proof.

\[
\begin{align*}
\{ & \text{PROP}() \; \text{LOCAL}(\text{temp } t; \text{temp } u) \\
& \text{SEP}(t \mapsto a; t + 4 \mapsto u; \text{listrep}(u, \text{cons}(b, l'''))) \\
\} \\
\{ & \text{PROP}() \; \text{LOCAL}(\text{temp } t; \text{temp } u) \\
& \text{SEP}(t \mapsto a; t + 4 \mapsto u; \exists p. u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l''')) \\
\}\end{align*}
\]

Given \( p \).

\[
\begin{align*}
\{ & \text{PROP}() \; \text{LOCAL}(\text{temp } t; \text{temp } u) \\
& \text{SEP}(t \mapsto a; t + 4 \mapsto u; u \mapsto b; u + 4 \mapsto p; \text{listrep}(p, l''')) \\
& \begin{array}{l}
t = u; u = t \rightarrow \text{tail}; \\
\end{array} \\
\{ & \text{PROP}() \; \text{LOCAL}(\text{temp } t; \text{temp } u) \\
& \text{SEP}(t \mapsto a; t + 4 \mapsto u; u \mapsto b; u + 4 \mapsto p; \text{listrep}(p, l''')) \\
\}\end{align*}
\]

\footnote{In the original proof in section 6.2, we first derive that \( l''' \) is not nil from the fact that \( u \neq \text{null} \) and \text{listrep}(u, l'''). Thus there exists \( b \) and \( l''' \) such that \( l''' = \text{cons}(b, l''') \). Then, we know that \text{listrep}(u, l''') \) is equivalent to \( \exists p. u \mapsto b \ast u + 4 \mapsto p \). The proof segment here only poses a simpler version. Also, I turn original assertions into canonical form and omit some unrelated conjuncts here for conciseness.}
Here, the first step is to unfold the recursive definition of listrep($\cdot$, $\cdot$), replacing listrep($u$, cons($b$, $l''$)) with:

$$\exists p. \ u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l''')$$

For later proofs, we would like to extract the existential variable $p$ and flatten the SEP clauses. This can be done by one tactic: Intros $p$.

It is a common scenario that a recursively defined separation logic predicate (for example, the predicate for linked list, for binary search tree, etc) is replaced with an existentially quantified separating conjunction. Flattening the SEP clauses is designed as one effect of Intros, so that users can extract existentials and improve canonical form arrangement in one tactic.

13.6.4 Discussion: are extraction rules necessary?

Extracting existentials and pure propositions is not necessary in writing decorated programs. For example, the following two proofs verify the same piece of C code (as the example in the previous section).

\[
\begin{align*}
\begin{array}{l}
\begin{aligned}
\text{PROP}() & \text{ LOCAL}(\text{temp } t; \text{temp } u \ u) \\
\text{SEP}(t \mapsto a; \ t + 4 \mapsto u; \ \text{listrep}(u, \text{cons}(b, l''')))
\end{aligned}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\begin{aligned}
\text{PROP}() & \text{ LOCAL}(\text{temp } t; \text{temp } u \ u) \\
\text{SEP}(t \mapsto a; \ t + 4 \mapsto u; \ \exists p. \ u \mapsto b \ast u + 4 \mapsto p \ast \text{listrep}(p, l'''))
\end{aligned}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\begin{aligned}
\text{PROP}() & \text{ LOCAL}(\text{temp } t; \text{temp } u \ u) \\
\text{SEP}(t \mapsto a; \ t + 4 \mapsto u; \ u \mapsto b; \ u + 4 \mapsto p; \ \text{listrep}(p, l'''))
\end{aligned}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\begin{aligned}
t = u;
\end{aligned}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\begin{aligned}
\text{PROP}() & \text{ LOCAL}(\text{temp } t; \text{temp } u \ u) \\
\text{SEP}(t \mapsto a; \ t + 4 \mapsto u; \ u \mapsto b; \ u + 4 \mapsto p; \ \text{listrep}(p, l'''))
\end{aligned}
\end{array}
\end{align*}
\]
\begin{align*}
\text{u} &= \text{t} \rightarrow \text{tail}; \\
\{ & \{ \text{PROP()} \ \text{LOCAL}(\text{temp t u}; \text{temp u p}) \\
& \ \ \ \text{SEP}(\text{t} \mapsto a; \ \text{t} + 4 \mapsto \text{u}; \ \text{u} \mapsto b; \ \text{u} + 4 \mapsto \text{p}; \ \text{listrep}(\text{p}, l''') \} \\
& \ \ \ \text{SEP}(\text{t} \mapsto a; \ \text{t} + 4 \mapsto \text{u}; \ \text{u} \mapsto b; \ \text{u} + 4 \mapsto \text{p}; \ \text{listrep}(\text{p}, l''') \} \\
& \ \ \ \text{PROP()} \ \text{LOCAL}(\text{temp t t}; \text{temp u u}) \\
& \ \ \ \text{SEP}(\text{t} \mapsto a; \ t + 4 \mapsto \text{u}; \text{∃p. u} \mapsto b * \text{u} + 4 \mapsto \text{p} * \text{listrep}(\text{p}, l''') \} \\
& \ \ \ \text{∃p. PROP()} \ \text{LOCAL}(\text{temp t t}; \text{temp u u}) \\
& \ \ \ \text{SEP}(\text{t} \mapsto a; \ t + 4 \mapsto \text{u}; \text{u} \mapsto b; \ u + 4 \mapsto \text{p}; \ \text{listrep}(\text{p}, l''') \} \\
\} \end{align*}

We can see that these two proofs well correspond to each other. The only inconvenience of the latter version is that we have to repeat the existential variable in multiple assertions—it seems that the extraction rules are unnecessary. Current VST-Floyd does not support the latter proof because the forward requires preconditions to be canonical. But this is not a major challenge, we could prove another set of derived rules like semax-load-canon and build corresponding tactics to derive strongest postconditions of assignment commands from existentially quantified canonical preconditions.

Here is the real problem of not having hoare-pure and hoare-exist in formalizing Hoare logic proofs in Coq. Once a variable or a pure proposition is extracted
from the precondition and put into Coq’s assumptions, users can use Coq’s built-in tactics to derive more properties in assumptions and rewrite in the conclusion. Coq is a more than 30-year-old tool and these built-in tactics are very convenient to use. If variables and assumptions were buried as existentials and PROP clauses in preconditions, such Coq built-ins would become inapplicable.

13.7 Introducing assumptions that are derivable from preconditions

I have shown that Intros can extract propositions from PROP clauses. Besides that, VST-Floyd provides another tactic assert-PROP which adds a proposition $P_0$ into the Coq assumption if it is derivable from the precondition.

Fig. 13.16 shows an example of assert-PROP. Here, (isptr $p$) says $p$ is a pointer value and it is not null. Applying assert-PROP (isptr $p$) adds this proposition to the Coq assumption list (see the change from the proof goal on the top right to the one on the bottom right). The users must then prove a separation logic entailment: precondition implies (isptr $p$). We know this is true because the SEP clauses show that some data is stored at address $p$.

The soundness of assert-PROP is ensured by the following Coq theorem:

\[ \text{assert-prop} : \forall \Delta P \text{ Pure} \] 
\[ P \vdash P^{\text{Pure}} \text{ and} \] 
\[ \text{assert-prop: } \Sigma; \Gamma; P^{\text{Pure}}; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \} \] 
\[ \text{then } \Sigma; \Gamma; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \}. \]

**Lemma** assert-PROP: for all $\Delta P^{\text{Pure}} P c Post$,

\[ P \vdash \llbracket P^{\text{Pure}} \rightarrow \] 
\[ (P^{\text{Pure}} \rightarrow \text{semax} \Delta P c Post) \rightarrow \] 
\[ \text{semax} \Delta P c Post. \]


\[ \Delta \vdash \begin{cases} \text{PROP}() \\ \text{LOCAL}(\text{temp} \times p) \\ \text{SEP}(p \rightarrow int a) \end{cases} \]

COMMANDS
\{ \text{POSTCONDITION} \}

\[ \Delta \wedge \begin{cases} \text{PROP}() \\ \text{LOCAL}(\text{temp} \times p) \\ \text{SEP}(p \rightarrow int a) \end{cases} \]

\[ \vdash \neg!\left( \text{isptr} \ p \right) \]

\[ H : \text{isptr} \ p \]

\[ \Delta \vdash \begin{cases} \text{PROP}() \\ \text{LOCAL}(\text{temp} \times p) \\ \text{SEP}(p \rightarrow int a) \end{cases} \]

COMMANDS
\{ \text{POSTCONDITIONS} \}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example}
\caption{Example: assert\_PROP (isptr \ p)}
\end{figure}

\section{13.8 Summary}

In summary, VST-Floyd provides four kinds of tactics for program verification: (1) tactics for verifying singleton C commands (\textit{forward} and \textit{forward-call}), (2) tactics for verifying structured C commands (\textit{forward-if} and \textit{forward-loop}), (3) tactics for weakening preconditions (\textit{gather-SEP} and \textit{replace-SEP}) and (4) the \textit{Intros} tactic for extracting existentially quantified variable and program state irrelevant propositions (i.e. pure assertions).

The technique that we used to build the tactic library is not C-language specific, and parts of it are not even separation logic specific. Here is a list—I underscore the
separation-logic-specific parts and double underscore the Verifiable-C-specific parts:

1. The soundness of forward.if and forward.loop is based on SEMAX-IF, SEMAX-LOOP and HOARE-SEQ. The Hoare logic rules for if rules and loop rules in different imperative languages are similar.

Forward.if also extracts the if condition (being true or being false) to the Coq context. That is based on HOARE-PURE.

2. The soundness of gather.SEP and replace.SEP is based on HOARE-CON, *-ASSOC and SEPCON-MONO (see Chapter 2). We could develop a more general (but less convenient) tactic for weakening preconditions instead—that is, to let users offer a full assertion instead of a small conjunct. This alternative tactic would only need the consequence rule HOARE-CON.

3. The soundness of Intros is mainly based on HOARE-EXISTS and HOARE-PURE. As a minor effect, sometimes users can also use Intros to pull out existential quantifiers from internal clauses first. That is sound due to HOARE-CON and the commutativity between the existential quantifier and the separating conjunction.

4. The implementation of forward and forward.call is the most complicated. It is based on the sequence rule (HOARE-SEQ), the consequence rule (HOARE-CON) and verified strongest postcondition generators which apply carefully designed proof rules (which are derivable from Verifiable C’s separation Hoare logic). More concretely, only the internal implementation of our strongest postcondition generators is C specific and separation logic specific. Calling it from forward and forward.call is not.

Besides offering these tactics to apply the program logic, it is also important for VST-Floyd to allow users to use Coq’s built-in tactics as much as possible. Because
of this principle, it would be more preferable to put assumptions in the Coq context instead of in preconditions. For example, instead of saying $[x] \leq [y]$ in the precondition, we would rather write:

$$\exists x, y. [x] = x \land [y] = y \land x \leq y.$$ 

Then the existentially quantified variables $x$ and $y$ and the proposition $x \leq y$ can be extracted from the precondition to the Coq context. It is the canonical form which makes sure that program variables are always isolated like this in preconditions.

Also, the canonical form plays an important role in building efficient (computational) strongest postcondition generators (see Chapter 12).
Chapter 14

Forward verification tactics by nonderivable proof rules

In the previous chapter, I introduced VST-Floyd’s tactics for forward verification. These tactics apply Hoare logic proof rules in a well-designed order so that the result of symbolic execution is presented to VST-Floyd’s users as reduced Coq proof goals. All Hoare logic proof rules used in these tactics are derived rules of Verifiable C’s separation Hoare logic (chapter 2).

In 2018, the VST team made two improvements to `forward_if` and `forward_loop`. But these improvements are based on new Hoare logic rules that are not derivable from Verifiable C’s original set of proof rules. For example:

\[
\text{SEMAX-IF-SEQ:}
\]

If \( \Sigma; \Gamma; P \vdash b \Downarrow \),

\[
\Sigma; \Gamma; \Delta \vdash \{ \llbracket b \rrbracket = \text{true} \wedge P \} \ c_1; c_3 \ Q, [R] \quad \text{and}
\]

\[
\Sigma; \Gamma; \Delta \vdash \{ \llbracket b \rrbracket = \text{false} \wedge P \} \ c_2; c_3 \ Q, [R]
\]

then \( \Sigma; \Gamma; \Delta \vdash \{ P \} \ 	ext{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2; \ c_3 \ Q, [R] \).

These new rules were proved sound in Coq by Andrew W. Appel, an expert of step indexed semantics and Verifiable C’s proof strategy. Proving new rules sound
w.r.t. to the semantics directly is not an ideal solution. It would be better if we can
derive them from the original rules.

In this chapter, I introduce these two improvements, explain the difficulty of deriving them from primary Hoare logic proof rules and discuss potential better solutions.

## 14.1 Improving **forward_if**

The original version of **forward_if** requires users to provide the normal postcondition (break condition, continue condition and return condition are not needed) of whole if command when that if command is not the last command in a basic block. The reason for this design is that we have no good automatic approach to derive a (possibly existentially quantified) canonical joint postcondition from the disjunction of two branches’ postconditions.

But consider the following sample programs:

```
// sample 1
if (x == 0)
    return 0;
MORE_COMMANDS
```
/ sample 2 (appears inside a loop)
if (x != 0)
s = s + x;
else {
    if (y != 0)
        break;
    else
        continue;
}
MORE_COMMANDS

// sample 3
if (flag)
    return 0;
else {
p = (int *) surely_malloc(sizeof(int));
* p = 1;
}
MORE_COMMANDS

In all of these examples, one branch of the if command will never terminate normally; it will always break, continue or return. In sample 1 and sample 3, the if-then branch will always return. In sample 2, the if-else branch will either break or continue. That means, the normal postconditions of these branches should be false and the joint postconditions of those if commands should be equivalent to the normal postcondition of the opposite branches. For such situations, forward-if can be improved so that users do not need to provide a joint postcondition.
A naive idea for implementing this improvement is to “\texttt{eapply semax\_seq}” at the beginning of \texttt{forward\_if} and \texttt{apply} one of the following proof rules afterwards:

\textsc{semax-if-then-canon}:

If $\Sigma; \Gamma; \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \vdash b \downarrow,$

\[
\text{msubst\_eval\_expr}(b, \overrightarrow{Q}) = \text{Some}(v),
\]

$\Sigma; \Gamma; \text{typed\_true}(v); \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ $c_1 \{ S, [\overrightarrow{T}] \}$ and

$\Sigma; \Gamma; \text{typed\_false}(v); \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ $c_2 \{ \bot, [\overrightarrow{T}] \}$

then $\Sigma; \Gamma; \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ if $b$ then $c_1$ else $c_2 \{ S, [\overrightarrow{T}] \}$.

\textsc{semax-if-else-canon}:

If $\Sigma; \Gamma; \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \vdash b \downarrow,$

\[
\text{msubst\_eval\_expr}(b, \overrightarrow{Q}) = \text{Some}(v),
\]

$\Sigma; \Gamma; \text{typed\_true}(v); \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ $c_1 \{ \bot, [\overrightarrow{T}] \}$ and

$\Sigma; \Gamma; \text{typed\_false}(v); \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ $c_2 \{ S, [\overrightarrow{T}] \}$

then $\Sigma; \Gamma; \Delta \vdash \{ \text{PROP}() \text{LOCAL} \overrightarrow{Q} \text{SEP} \overrightarrow{R} \}$ if $b$ then $c_1$ else $c_2 \{ S, [\overrightarrow{T}] \}$.

The idea is, after applying \textsc{semax-if-then-canon}, the verification of the if-then branch will instantiate the joint postcondition. And similarly, after applying \textsc{semax-if-else-canon}, the verification of the if-else branch will instantiate the joint postcondition. As a result, users do not need to manually supply that postcondition.

However, this new implementation of \texttt{forward\_if} does not work: when the if-then branch does always \texttt{break}, \texttt{continue} or \texttt{return}, the if-else branch’s postcondition (which is generated naturally using VST-Floyd) may contain newly introduced variables. The sample program 3 above is such an example. Here is a forward verification of its if-else branch:
Assume \( b = \text{false} \)

\[
\{ \text{PROP()} \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b))) \ \text{SEP()} \}
\]

\[
p = (\text{int *}) \text{surely_malloc(sizeof(int))};
\]

\[
\{ \exists p. \ \text{PROP()} \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b))); \text{temp p p} \ \text{SEP}(p \mapsto \_)} \}
\]

Given \( p \),

\[
\{ \ \text{PROP()} \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b))); \text{temp p p} \ \text{SEP}(p \mapsto \_)} \}
\]

\[
* \ p = -1;
\]

\[
\{ \ \text{PROP()} \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b)); \text{temp p p} \ \text{SEP}(p \mapsto -1)} \}
\]

Here, since \( p \) is only introduced into the logical context in the if-else branch’s verification process, it is illegal to appear in the joint postcondition. The correct joint postcondition should be:

\[
\{ \exists p. \ \text{PROP}(b = \text{false}) \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b)); \text{temp p p} \ \text{SEP}(p \mapsto -1)} \}
\]

Suppose that the verification of the if command in sample program 3 above starts with logical context \( \Sigma \) and assumptions \( \Gamma \). The context and assumptions become \( \Sigma; p \) and \( \Gamma; b = \text{false} \) at the end of verification of that if-else branch. One possible workaround would be to force users to revert that extra logical variable and assumption back, generate the joint postcondition and extract them when starting to verify the C commands after if-then-else. But that is very inconvenient. It would be much better if users can directly start verifying their later code with context \( \Sigma; p \), assumptions \( \Gamma; b = \text{false} \) and precondition

\[
\{ \ \text{PROP()} \ \text{LOCAL}(\text{temp flag (Val.of_bool}(b)); \text{temp p p} \ \text{SEP}(p \mapsto -1)} \}. \]
Implementation. When forward-if is called without a parameter and the if command is not the last command in the basic block, we will first do a trivial syntactic analysis to make sure that at least one branch of that if command will always break, continue or return. After that, forward-if will apply SE MAX-IF-SEQ-CANON (the canonical form version of SE MAX-IF-SEQ):

Lemma SE MAX-IF-SEQ-CANON: forall Δ \rightarrowawning Q \rightarrowawning R v b c_1 c_2 c_3 Post, PROP () LOCAL \rightarrowawning Q \ SE P \rightarrowawning R \vdash tc_{expr} Δ b \rightarrow.
msubst_{eval_{expr}} b \rightarrowawning Q = Some v \rightarrow.
(typed_{true} v \rightarrow \text{semax} Δ (PROP () LOCAL \rightarrowawning Q \ SE P \rightarrowawning R) (c_1; c_3) \text{Post}) \rightarrow.
(typed_{false} v \rightarrow \text{semax} Δ (PROP () LOCAL \rightarrowawning Q \ SE P \rightarrowawning R) (c_2; c_3) \text{Post}) \rightarrow.
\text{semax} Δ (PROP () LOCAL \rightarrowawning Q \ SE P \rightarrowawning R) (if b then c_1 else c_2; c_3) \text{Post}.

The first two subgoals (corresponding to the first two assumptions in this proof rule) will be solved automatically. The last two subgoals will be left to the users (after improving and intro the typed_{true}/typed_{false} expression).

Why does this implementation work? Suppose that the if-then branch will always break, continue or return. Users’ verification of

$$\Sigma; \Gamma; v \text{ is true}; Δ \vdash \{\text{PROP()} \ LOCAL \rightarrowawning Q \ SE P \rightarrowawning R\} c_1; c_3 \{\text{Post}\}$$

will not touch \text{c}_3 and the verification will always end with those break, continue and return commands in \text{c}_1. Moreover, users’ verification of the if-else branch and the code afterwards is just to prove:

$$\Sigma; \Gamma; v \text{ is false}; Δ \vdash \{\text{PROP()} \ LOCAL \rightarrowawning Q \ SE P \rightarrowawning R\} c_2; c_3 \{\text{Post}\}$$

The new logical variables and assumptions introduced during the verification process of \text{c}_2 can be directly used in the verification of \text{c}_3.
14.2 Improving forward_loop

One annoying thing about verifying a loop is that users have to provide three assertions: two loop invariants (one before the loop body and one before the increment step) and one postcondition.

There could be some improvement towards this setting. For example, when the increment command is just a skip (Clight’s formalization of while loop is an example this situation), the loop invariants should be identical.

The improvement that I will introduce in this section is for those loops that no continue command is involved in the loop body. Consider the following C program, which is just a variant of the linked-list-append function that I verified in chapter 6.

```c
struct list {int head; struct list *tail;};

struct list * append1 (struct list * x, struct list * y) {
    struct list * t, * u;
    if (x == NULL)
        return y;
    return x;
    else {
        \{ [x] = x ∧ [y] = y ∧ l_1 = cons(a, l'_1) ∧
        x ↦→ h * x + 4 ↦→ p * listrep(p, l'_1) * listrep(y, l_2)
        t = x; u = t → tail;
        for (; u != NULL; u = t → tail) t = u;
        t → tail = y;
        \{ [x] = x ∧ listrep(x, l_1 · l_2)\}
        return x;
    }
}
```

In this program, a for loop is used instead of a while loop. According CompCert Clight’s formalization, the loop body is:

```
```

178
if (u != NULL) break; t = u;

the increment step is:

\[ u = t \rightarrow \text{tail}; \]

In this case, the loop body does not have any continue command inside.

If verifying it using the original forward loop, users have to provide two loop invariants. The first invariant (the one before the loop body) is:

\[ \exists a, l_1', l_1'', t, u. \]

\[ \text{PROP}(l_1 = l_1' \cdot \text{cons}(a, l_1'')) \]

\[ \text{LOCAL}(\text{temp } x; \text{temp } y; \text{temp } t; \text{temp } u; u) \]

\[ \text{SEP}(t \mapsto a, u; \text{listrep}(u, l_1''); \text{listrep}(y, l_2); \text{lseg}(x, t, l_1')) \]

The second invariant (the one before the increment step) is:

\[ \exists b, l_1', l_1'', u, p. \]

\[ \text{PROP}(l_1 = l_1' \cdot \text{cons}(b, l_1'')) \]

\[ \text{LOCAL}(\text{temp } x; \text{temp } y; \text{temp } u; \text{temp } u; u) \]

\[ \text{SEP}(u \mapsto b, p; \text{listrep}(p, l_1''); \text{listrep}(y, l_2); \text{lseg}(x, u, l_1')) \]

Users need to extract existentials at the beginning and instantiate existentials at the end both when verifying the loop body and when verifying the increment step. This is inconvenient.

In general situations, an execution can reach the increment step either after finishing the loop body’s execution normally or when reaching a continue in the loop body.

Since the loop body in this example does not contain a continue, it is enough to verify this loop by showing that the loop body together with the increment step
preserve the first loop invariant. In other words,

\[
\exists a, l_1', l_1'', t, u. \\
\text{PROP}(l_1 = l_1' \cdot \text{cons}(a, l_1'')) \\
\text{LOCAL}(\text{temp } x; \text{temp } y; \text{temp } t; \text{temp } u; u) \\
\text{SEP}(t \overset{\text{int}}{\rightarrow} a, u; \text{listrep}(u, l_1''); \text{listrep}(y, l_2); \text{lseg}(x, t, l_1'))
\]

\[
\quad \text{if}(u! = \text{NULL})\text{break}; t = u; u = u \rightarrow \text{tail;}
\]

Then, users need to extract and instantiate existentials only once.

The improved forward_loop tests whether the loop body contains a continue or not. If not, it allows users not to provide the second loop invariant (which was the original continue assertion) and applies the following proof rule:

**Lemma SEMAX_LOOP_NOCONTINUE:**forall $\Delta c c_i P I Q Q_{brk} Q_{con} Q_{ret}$,

\[
P \vdash I \rightarrow \\
\text{semax } \Delta (c; c_i) (I, [Q, \bot, Q_{ret}]) \rightarrow \\
\text{semax } \Delta P \text{ loop } (c_i) c (Q, [Q_{brk}, Q_{con}, Q_{ret}]).
\]

### 14.3 The difficulty of deriving new proof rules

The reason that I put these two tactic improvements separately in this section is that they both use nonderivable Hoare logic proof rules—SEMAX-IF-SEQ and SEMAX-LOOP-NOCONTINUE. The forward verification tactics that I introduced in the last chapter are based on many Hoare logic proof rules. No matter how complicated they
are, those rules are all derived from Verifiable C’s separation Hoare logic, i.e. the Hoare logic rules that I presented in chapter 2. For example, SEMAX-LOAD-CANON is derivable from SEMAX-LOAD and HOARE-CON.

Why are SEMAX-IF-SEQ and SEMAX-LOOP-NOCONTINUE not derivable? One one hand, it is because Verifiable C only provides a unique proof rule for each different compound command. For example, the Hoare triple of a sequence of two commands can only be proved by HOARE-SEQ (HOARE-EXISTS, HOARE-PURE and HOARE-CON may be used as glue steps); the Hoare triple of an if command can only be proved by SEMAX-IF.

As a result, to derive SEMAX-IF-SEQ:

SEMEX-IF-SEQ:

If \( \Sigma; \Gamma; P \vdash b \downarrow \),

\[
\Sigma; \Gamma; \Delta \vdash \{ [b] = \text{true} \land P \} \ c_1; c_3 \ \{ Q', [R] \} \text{ and }
\]

\[
\Sigma; \Gamma; \Delta \vdash \{ [b] = \text{false} \land P \} \ c_2; c_3 \ \{ Q, [R] \}
\]
then \( \Sigma; \Gamma; \Delta \vdash \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2; \ c_3 \ \{ Q, [R] \} \).

One has to find an assertion \( Q' \), such that the following triples can be proved from SEMAX-IF-SEQ’s assumptions:

\[
\Sigma; \Gamma; \Delta \vdash \{ [b] = \text{true} \land P \} \ c_1 \ \{ Q', [R] \}
\]

\[
\Sigma; \Gamma; \Delta \vdash \{ [b] = \text{false} \land P \} \ c_2 \ \{ Q', [R] \}
\]

\[
\Sigma; \Gamma; \Delta \vdash \{ Q' \} \ c_3 \ \{ Q, [R] \}
\]

On the other hand, Verifiable C’s proof rules cannot decompose a Hoare triple into Hoare triples of subprograms. Thus, deriving the triples above from SEMAX-IF-SEQ’s assumptions is impossible.

A similar difficulty shows up when trying to derive SEMAX-LOOP-NOCONTINUE. That is, from \( \Sigma; \Gamma; \Delta \vdash \{ I \} \ c; c_i \ \{ I \} \), one cannot prove the existence of \( I_{\text{con}} \) such that

\[
\Sigma; \Gamma; \Delta \vdash \{ I \} \ c \ \{ I_{\text{con}} \} \quad \Sigma; \Gamma; \Delta \vdash \{ I_{\text{con}} \} \ c_i \ \{ I \} .
\]
Remark 1. Does Verifiable C choose a wrong proof rule set so that these two new rules are not derivable? The answer is no. Verifiable C follows the tradition of Hoare logics—providing proof rules for every compound C command. Cook \cite{22} proved that such Hoare logics are relatively complete. In other words, if the proof rules for assertion entailments are powerful enough to derive all valid entailments, then the Hoare logic is powerful enough to derive all valid Hoare triples. In the case of \texttt{SEMAX-IF-SEQ} and \texttt{SEMAX-LOOP-NOCONTINUE}, all of their concrete instances are actually derivable by Verifiable C.

Remark 2. How bad is it that these two rules are not derivable? Currently, these new rules are proved sound directly with respect to CompCert Clight’s operational semantics and Verifiable C’s semantic definition of Hoare triple’s validity. \texttt{SEMAX-IF-SEQ}’s soundness takes an expert of Verifiable C one day to prove in Coq and \texttt{SEMAX-LOOP-NOCONTINUE} takes more than one week. In comparison, the fact that proof rules like \texttt{SEMAX-SEQ}’ are derivable is obvious for any one who knows Hoare logic and does not know Verifiable C’s semantic definition. Coq proofs will only take minutes.

14.4 Solution 1: proof rules of weakest precondition

The first potential solution for Verifiable C is to provide Hoare logic proof rules in a different form. Instead of providing proof rules for constructing Hoare triples, Verifiable C could present those proof rules as properties of weakest preconditions.

Suppose \( \text{wp}_c^\Delta (Q; [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]) \) represents the weakest precondition under assumption \( \Delta \) that can ensure the safe execution of \( c \) and ensure that the ending state will satisfy \( Q, Q_{\text{brk}}, Q_{\text{con}} \) or \( Q_{\text{ret}} \), if that execution terminates normally, by \texttt{break}, by
continue or by return respectively. Then it is well known that:

\[ \text{wp-spec : } \Sigma; \Gamma; \Delta \vdash \{ P \} c \left\{ Q, [\vec{R}] \right\} \iff \Sigma; \Gamma; P \vdash \wp^c_\Delta \left( Q, [\vec{R}] \right) \]

Iris \cite{iris}, another separation logic tool for interactive verification, chooses to formalize their proof rules through weakest precondition.

For example, the sequence rule can be restated as follows:

\[ \text{wp-seq : } \Sigma; \Gamma; \wp^\Delta_{c_1,c_2} \left( Q, [\vec{R}] \right) \vdash \wp^\Delta_{c_1} \left( \wp^\Delta_{c_2} \left( Q, [\vec{R}] \right), [\vec{R}] \right) \]

\[ \text{wp-seq is actually stronger than hoare-seq.} \]

**Theorem 8.** The following proof rule is derivable from wp-seq and wp-spec:

**HOARE-SEQ-INV:**

\[
\text{If } \Sigma; \Gamma; \Delta \vdash \{ P \} c_1; c_2 \left\{ R, [\vec{S}] \right\} \\
\text{then } \Sigma; \Gamma; \Delta \vdash \{ P \} c_1 \left\{ \wp^\Delta_{c_2} \left( R, [\vec{S}] \right), [\vec{S}] \right\}.
\]

**Proof.** By wp-spec and the assumption,

\[ \Sigma; \Gamma; P \vdash \wp^\Delta_{c_1,c_2} \left( R, [\vec{S}] \right) \]

By wp-seq, \[ \Sigma; \Gamma; P \vdash \wp^\Delta_{c_1} \left( \wp^\Delta_{c_2} \left( R, [\vec{S}] \right), [\vec{S}] \right) \]

By wp-spec again, \[ \Sigma; \Gamma; \Delta \vdash \{ P \} c_1 \left\{ \wp^\Delta_{c_2} \left( R, [\vec{S}] \right), [\vec{S}] \right\} \]

This theorem means that with proof rule wp-seq and wp-spec,

\[ \Sigma; \Gamma; \Delta \vdash \{ P \} c_1; c_2 \left\{ R, [\vec{S}] \right\} \]

if and only if there exists an assertion \( Q \) such that \( \Sigma; \Gamma; \Delta \vdash \{ P \} c_1 \left\{ Q, [\vec{S}] \right\} \) and \( \Sigma; \Gamma; \Delta \vdash \{ Q \} c_2 \left\{ R, [\vec{S}] \right\} \). Formally, this can be stated as the following corollary.
Corollary 9. The following proof rule is derivable from \( \text{wp-seq} \) and \( \text{wp-spec} \):

**HOARE-SEQ-INV’:**

If \( \Sigma; \Gamma; \Delta \vdash \{P\} c_1; c_2 \left\{ R, [\vec{S}] \right\} \)
then \( \Sigma; \Gamma; \Delta \vdash \{P\} c_1 \left\{ Q, [\vec{S}] \right\} \) and
\( \Sigma; \Gamma; \Delta \vdash \{Q\} c_2 \left\{ R, [\vec{S}] \right\} \)
in which \( Q \triangleq \text{wp}_{c_2}^\Delta \left( R, [\vec{S}] \right) \).

The following theorems’ proofs derive \( \text{semax-if-seq} \) and \( \text{semax-loop-nocontinue} \).

**Theorem 10.** \( \text{semax-if-seq} \) is derivable from \( \text{semax-if} \), \( \text{wp-seq} \) and \( \text{wp-spec} \).

**Proof.** Suppose

\[
\Sigma; \Gamma; P \vdash b \Downarrow
\]
\[
\Sigma; \Gamma; \Delta \vdash \{[b] = \text{true} \land P\} c_1; c_3 \left\{ Q, [\vec{R}] \right\}
\]
\[
\Sigma; \Gamma; \Delta \vdash \{[b] = \text{false} \land P\} c_2; c_3 \left\{ Q, [\vec{R}] \right\}.
\]

We know that

\[
\Sigma; \Gamma; \Delta \vdash \{[b] = \text{true} \land P\} c_1 \left\{ \text{wp}_{c_3}^\Delta \left( Q, [\vec{R}] \right), [\vec{R}] \right\}
\]
\[
\Sigma; \Gamma; \Delta \vdash \{[b] = \text{false} \land P\} c_2 \left\{ \text{wp}_{c_3}^\Delta \left( Q, [\vec{R}] \right), [\vec{R}] \right\}
\]

by \( \text{HOARE-SEQ-INV} \). Thus, by \( \text{semax-if} \):

\[
\Sigma; \Gamma; \Delta \vdash \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2 \left\{ \text{wp}_{c_3}^\Delta \left( Q, [\vec{R}] \right), [\vec{R}] \right\}
\]

By \( \text{wp-spec} \), \( \Sigma; \Gamma; P \vdash \text{wp}_{c_2}^\Delta b \text{ then } c_1 \text{ else } c_2 \left( \text{wp}_{c_3}^\Delta \left( Q, [\vec{R}] \right), [\vec{R}] \right) \).

By \( \text{wp-seq} \), \( \Sigma; \Gamma; P \vdash \text{wp}_{c_2}^\Delta b \text{ then } c_1 \text{ else } c_2; c_3 \left( Q, [\vec{R}] \right) \).

By \( \text{wp-spec again} \), \( \Sigma; \Gamma; \Delta \vdash \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2; c_3 \left\{ Q, [\vec{R}] \right\} \).
Theorem 11. \textsc{semax-loop-nocontinue} is derivable from \textsc{semax-loop}, \textsc{hoare-con}, \textsc{wp-seq} and \textsc{wp-spec}.

Proof. Suppose \(\Sigma; \Gamma; \Delta \vdash \{P\} \ c; c_i \ \{P, [Q, \bot, R]\}\).

By \textsc{hoare-seq-inv}, \(\Sigma; \Gamma; \Delta \vdash \{P\} \ c \ \{wp_{ci}^\Delta (P, \{Q, \bot, R\}), \{Q, \bot, R\}\}\).

By \textsc{hoare-con}, \(\Sigma; \Gamma; \Delta \vdash \{P\} \ c \ \{wp_{ci}^\Delta (P, \{Q, \bot, R\}), \{Q, wp_{ci}^\Delta (P, \{Q, \bot, R\}), R\}\}\).

Since \(\Sigma; \Gamma; wp_{ci}^\Delta (P, \{Q, \bot, R\}) \vdash wp_{ci}^\Delta (P, \{Q, \bot, R\})\), we know by \textsc{wp-spec} that
\[
\Sigma; \Gamma; \Delta \vdash \{wp_{ci}^\Delta (P, \{Q, \bot, R\})\} \ c_i \ \{P, [Q, \bot, R]\}\.
\]

Thus, if taking \(P\) as the first loop invariant and taking \(wp_{ci}^\Delta (P, \{Q, \bot, R\})\) as the second loop invariant, we know by \textsc{semax-loop}:
\[
\Sigma; \Gamma; \Delta \vdash \{P\} \ \text{loop}(c_i) c \ \{Q, \{R_{brk}, R_{con}, R\}\} \quad \square
\]

These two theorems tell us, if we formalize “weakest precondition” and add \textsc{wp-spec} and \textsc{wp-seq} to Verifiable C’s proof rules, \textsc{semax-if-seq} and \textsc{semax-loop-nocontinue} becomes derivable. Also, \textsc{wp-spec} and \textsc{wp-seq} give us power to decompose a triple to get Hoare triples of its subprograms. That means, if VST-Floyd would need another proof rule that rearranges C commands like \textsc{semax-if-seq} does, it is derivable as well.

The only drawback of the solution is that we have to prove \textsc{wp-spec} and \textsc{wp-seq} sound first. Due to the complexity of Verifiable C’s semantic definition of Hoare triples, this soundness proof is very hard.\footnote{It is not known yet whether \textsc{wp-spec} and \textsc{wp-seq} can be proved sound in Verifiable C now. If we call “if the beginning state of C command \(c\) satisfies \(P_1\), executing \(c\) is safe and the ending state will satisfy \(P_2, P_3, P_4\) or \(P_5\) if the execution terminates normally,” by \textsc{break}, \textsc{continue} or \textsc{by return}” the normal semantic definition of \(\{P_1\} \ c \ \{P_2, [P_3, P_4, P_5]\}\). Verifiable C’s semantic definition \textsc{semax} is actually weaker than this normal semantic definition. I have attempt to formalize \(wp_{\{\}}\) using a correspondingly weaker definition. In that case, \textsc{wp-spec} is easily proved sound but I fail to prove \textsc{wp-seq}. If I turn to formalize \(wp_{\{\}}\) using a normal semantic definition, then \textsc{wp-spec} becomes a problem. Iris’s semantic definition of Hoare triples are normal, so they do not have this problem.}
14.5 Solution 2: Hoare logic as a proof theory

This section discusses another potential solution other than adding two stronger proof rules WP-SEQ and WP-SPEC.

Previously, I have shown that the main difficulty of deriving SEMAX-IF-SEQ and SEMAX-LOOP-NOCONTINUE is that Verifiable C’s proof rules cannot decompose a Hoare triple into triples of subprograms. The main idea here is to consider the Hoare triples that are derivable from a fixed set of proof rules only. For example, if \( \Sigma; \Gamma; \Delta \vdash \{ P \} c_1; c_2 \{ R, [\vec{S}] \} \) can only be derivable by HOARE-SEQ, then there must exist an assertion \( Q \) such that \( \Sigma; \Gamma; \Delta \vdash \{ P \} c_1 \{ Q, [\vec{S}] \} \) and \( \Sigma; \Gamma; \Delta \vdash \{ Q \} c_2 \{ R, [\vec{S}] \} \).

**Definition 12.** A Hoare triple \( \Sigma; \Gamma; \Delta \vdash \{ P \} c \{ Q, [\vec{R}] \} \) is derivable from the minimum separation Hoare logic if it is derivable using HOARE-SEQ, HOARE-CON, SEMAX-SET, SEMAX-LOAD, SEMAX-STORE, SEMAX-CALL, SEMAX-IF, SEMAX-LOOP and proof rules for separation logic entailments only. In short, \( \Sigma; \Gamma; \Delta \vdash_{\min} \{ P \} c \{ Q, [\vec{R}] \} \).

**Theorem 13.** If \( \Sigma; \Gamma; \Delta \vdash_{\min} \{ P \} c_1; c_2 \{ R, [\vec{S}] \} \) then there exists an assertion \( Q \) such that \( \Sigma; \Gamma; \Delta \vdash_{\min} \{ P \} c_1 \{ Q, [\vec{S}] \} \) and \( \Sigma; \Gamma; \Delta \vdash_{\min} \{ Q \} c_2 \{ R, [\vec{S}] \} \).

**Proof.** I prove it by induction over the proof tree of deriving \( \Sigma; \Gamma; \Delta \vdash \{ P \} c_1; c_2 \{ R, [\vec{S}] \} \) from the minimum separation Hoare logic.

Base step. If the last step in the proof applies HOARE-SEQ, then the conclusion is obviously true.
Inductive step. Suppose the last step in the proof applies $\text{hoare-con}$ and this last step is based on the following assumptions:

$$
\Sigma; \Gamma; \Delta \vdash_{\min} \{P'\} c_1; c_2 \left\{ R', [\vec{S}] \right\}
$$

$$
\Sigma; \Gamma; P \vdash P'
$$

$$
\Sigma; \Gamma; R' \vdash R
$$

For any $i, \Sigma; \Gamma; S'_i \vdash S_i$

By the induction hypothesis corresponding to the smaller proof tree $\Sigma; \Gamma; \Delta \vdash_{\min} \{P'\} c_1; c_2 \left\{ R', [\vec{S}] \right\}$, there must exists $Q$ such that $\Sigma; \Gamma; \Delta \vdash_{\min} \{Q\} c_1 \left\{ Q, [\vec{S}] \right\}$ and $\Sigma; \Gamma; \Delta \vdash_{\min} \{Q\} c_2 \left\{ R, [\vec{S}] \right\}$. Thus, by $\text{hoare-seq}$, $\Sigma; \Gamma; \Delta \vdash_{\min} \{Q\} c_1 \left\{ Q, [\vec{S}] \right\}$ and $\Sigma; \Gamma; \Delta \vdash_{\min} \{Q\} c_2 \left\{ R, [\vec{S}] \right\}$. $\square$

**Remark.** Different from corollary [9], this theorem does not derive a new Hoare logic proof rule. Instead, it proves a metaproperty of the minimum separation Hoare logic.

The minimum separation Hoare logic does not have extraction rules. We can only prove two more metaproperties as follows. (Their proofs are long and can be found in Coq development.)

**Theorem 14.** If $\Sigma; x; \Gamma; \Delta \vdash_{\min} \{P\} c \left\{ Q, [\vec{R}] \right\}$ and $x$ does not freely occur in $\Gamma, Q$ and $\vec{R}$, then $\Sigma; \Gamma; \Delta \vdash_{\min} \{\exists x. P\} c \left\{ Q, [\vec{R}] \right\}$.

**Theorem 15.** If $\Sigma; \Gamma; P^{\text{Pure}}, \Delta \vdash_{\min} \{P\} c \left\{ Q, [\vec{R}] \right\}$, then $\Sigma; \Gamma; \Delta \vdash_{\min} \{P^{\text{Pure}} \land P\} c \left\{ Q, [\vec{R}] \right\}$.

By theorem [14] and

$$
\Sigma; \Gamma; P_1 \lor P_2 \vdash_{\min} \exists b : \text{bool}. \text{if } b \text{ then } P_1 \text{ else } P_2
$$

$\text{It is obvious that the last step in the proof applies either } \text{hoare-seq or } \text{hoare-con}$. 
we can immediately get:

**Corollary 16.** If \( \Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P_1 \} c \{ Q, [\vec{R}] \} \) and \( \Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P_2 \} c \{ Q, [\vec{R}] \} \), then \( \Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P_1 \lor P_2 \} c \{ Q, [\vec{R}] \} \).

Now, we are ready to prove metaproperties corresponding to \textsc{semax-if-seq} and \textsc{semax-loop-nocontinue}.

**Theorem 17.** If \( \Sigma; \Gamma; P \vdash b \downarrow \), \( \Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ \llbracket b \rrbracket = \text{true} \land P \} c_1; c_3 \{ Q, [\vec{R}] \} \) and \( \Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ \llbracket b \rrbracket = \text{false} \land P \} c_2; c_3 \{ Q, [\vec{R}] \} \), then

\[
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2; c_3 \{ Q, [\vec{R}] \}.
\]

**Proof.** According to theorem 13, there exists \( S_1 \) and \( S_2 \) such that

\[
\begin{align*}
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ \llbracket b \rrbracket = \text{true} \land P \} c_1 \{ S_1, [\vec{R}] \} \\
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ S_1 \} c_3 \{ Q, [\vec{R}] \} \\
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ \llbracket b \rrbracket = \text{true} \land P \} c_1 \{ S_2, [\vec{R}] \} \\
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ S_2 \} c_3 \{ Q, [\vec{R}] \}.
\end{align*}
\]

Thus, by \textsc{semax-if} and \textsc{hoare-con},

\[
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2; c_3 \{ S_1 \lor S_2, [\vec{R}] \}.
\]

On the other hand, by corollary 16,

\[
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ S_1 \lor S_2 \} c_3 \{ Q, [\vec{R}] \}.
\]

As a result of \textsc{hoare-seq},

\[
\Sigma; \Gamma; \Delta \vdash_{\text{min}} \{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2; c_3 \{ Q, [\vec{R}] \}. \quad \Box
\]
Theorem 18. If $\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} c; c_i \{P, [Q, \bot, R]\}$, then

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} \text{loop}(c_i)c \{Q, [R_{\text{brk}}, R_{\text{con}}, R]\}.$$ 

Proof. By theorem 13, there exists $P'$ such that,

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} c \{P', [Q, \bot, R]\}$$

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P'\} c_i \{P, [Q, \bot, R]\}$$

By Hoare-con,

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} c \{P', [Q, P', R]\}$$

Finally, by semax-loop:

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} \text{loop}(c_i)c \{Q, [R_{\text{brk}}, R_{\text{con}}, R]\}. \quad \square$$

This means: by replacing $\vdash$ with $\vdash_{\min}$, we will have the original set of proof rules and extra ones at metalogic level, e.g. semax-seq-if and semax-loop-nocontinue.

The concept of being derivable from the minimum separation Hoare logic only can be formalized in Coq as a Coq inductive proposition. From users’ point of view, using VST-Floyd to prove (*) and (**) are exactly the same:

$$\Sigma; \Gamma; \Delta \vdash \{P\} c \left\{Q, [\vec{R}]\right\} \quad (*)$$

$$\Sigma; \Gamma; \Delta \vdash_{\min} \{P\} c \left\{Q, [\vec{R}]\right\} \quad (**)$$

although they are two conceptually different tasks. Specifically, proving (*) via VST-Floyd means constructing a Hoare logic proof using Verifiable C’s proof rules. Proving (**) via VST-Floyd demonstrates that there exists a Hoare logic proof of (*) using
the minimum separation Hoare logic (a subset of Verifiable C’s proof rules), without constructing that proof directly. In the end, either (*) or (**) ensures: if the beginning state satisfies $P$ then executing $c$ is safe and the ending state will satisfy the corresponding postcondition ($Q$ or one in $\vec{R}$) if that execution terminates.
Chapter 15

Case study: BST insert verified in VST-Floyd

This chapter demonstrates the implementation correctness proof in VST for BST-insert as an example. The corresponding paper-and-pen proof is in section 7.3. Here is the C program.

```c
struct tree {int key; void *value;
    struct tree *left, *right;};

typedef struct tree **treebox;

void insert (treebox p, int x, void *v) {
    struct tree *q;
    while (1) {
        q = *p;
        if (q==NULL) {
            q = (struct tree *) surely_malloc (sizeof *p);
            q→key=x; q→value=v;
            q→left=NULL; q→right=NULL;
```
*p=q;
    return;
} else {
    int y = q→ key;
    if (x<y)
        p= &q→ left;
    else if (y<x)
        p= &q→ right;
    else {
        q→ value=v;
        return;
    }
}

Here is the specification:

**Definition** insert_spec :=

DECLARE .insert

WITH p0: val, x: nat, v: val, t0: tree val

PRE [ .p OF (tptr (tptr t -struct-tree)), .x OF tint,
    .value OF (tptr Tvoid) ]

PROP( Int.min-signed <= Z.of_nat x <= Int.max_signed; is_pointer_or_null v)

LOCAL(temp .p p0; temp .x (Vint (Int.repr (Z.of_nat x))); temp .value v)

SEP (treebox_rep t0 p0)

POST [ Tvoid ]

PROP()

LOCAL()

SEP (treebox_rep (ins t0 x v) p0).
Lemma body-insert: semax.body Vprog Gprog f.insert insert_spec.

Proof.

start_function.

forward_loop (EX p: val, EX t: tree val, EX P: tree val \rightarrow tree val,
PROP(P (insert x v t) = (insert x v t0))
LOCAL(temp _p p; temp _x (Vint (Int.repr (Z.of_nat x))); temp _value v)
SEP(treebox_rep t p; partial_treebox_rep P p0 p)).

(* Precondition *)
{
Exists p0 t0 (fun t: tree val \Rightarrow t). entailer!.
apply emp_partial_treebox_rep_H.
}

(* Loop body *)

Intros p t P.
rewrite treebox_rep.tree_rep at 1. Intros q.
forward. (* q = * p; *)
forward_if.
+ (* then clause *)
subst q.
forward_call (sizeof t.struct_tree).
{ simpl; rep_omega. }
Intros q.
rewrite memory_block_data_at_. by auto.
forward. (* q\rightarrow key=x; *)
forward. (* q\rightarrow value=value; *)
forward. (* q\rightarrow left=NULL; *)
forward. (∗ q→ right=NULL; ∗)
assert_PROP (t = (@N _)) by entailer!.
subst t. rewrite tree_rep.treebox_rep. normalize.
forward. (∗ ∗ p = q; ∗)
forward. (∗ return; ∗)
entailer!. clear -H1 H0 H.
sep_apply (treebox_rep.leaf x q p v); auto.
rewrite <- H1. change (T N x v N) with (insert x v N).
apply treebox_rep.partial_treebox_rep.
+ (∗ else clause ∗)
destruct t; rewrite tree_rep.treebox_rep.
{ normalize. }
Intros. clear H2.
forward. (∗ y=q→ key; ∗)
forward_if; [ | forward_if ].
-(∗ Inner if, then clause: x<k ∗)
forward. (∗ p=&q→ left ∗)
Exists (field_address t_struct_tree [StructField .left] q) t1 (fun t1 ⇒ P (T t1 k v0 t2)).
entailer!.
∗ rewrite <- H1.
simpl; simpl_compb; auto.
* sep_apply (partial_treebox_rep.singleton_left t2 k v0 q p); auto.
apply partial_treebox_rep.partial_treebox_rep.
-(∗ Inner if, second branch: k<x ∗)
forward. (∗ p=&q→ right ∗)
Exists (field_address t_struct_tree [StructField .right] q) t2 (fun t2 ⇒ P (T t1 k v0 t2)).
entailer!.   

194
* rewrite \(\langle-\ H1\). 

simpl; simpl\_compb; simpl\_compb; auto.

* sep\_apply (partial\_treebox\_rep\_singleton\_right t1 k v0 q p); auto.

cancel; apply partial\_treebox\_rep\_partial\_treebox\_rep.

\(-(*\ Inner\ if,\ third\ branch: x=k *)\)

assert (x=k) by omega.

subst x. clear H H2 H5.

forward. (* q\rightarrow value=value *)

forward. (* return *)

entailer!.

rewrite \(\langle-\ H1.

simpl insert. simpl\_compb; simpl\_compb.

sep\_apply (treebox\_rep\_internal t1 k v t2 p q); auto.

apply treebox\_rep\_partial\_treebox\_rep.

Qed.
A VST proof always starts with the \texttt{start\_function} tactic. In this proof, the proof goal after \texttt{start\_function} is:

\begin{verbatim}
p0 : val
x : nat
v : val
m0 : total\_map val
H : \texttt{Int.min\_signed} \leq \texttt{Z\_of\_nat} x \leq \texttt{Int.max\_signed}
H0 : \texttt{is\_pointer\_or\_null} v
t0 : tree val
Delta\_specs := abbreviate : \texttt{PTree.t funspec}
Delta := abbreviate : \texttt{tycontext}
LOOP := abbreviate : \texttt{statement}
POSTCONDITION := abbreviate : \texttt{ret\_assert}
\end{verbatim}

\begin{verbatim}
sem\_max Delta
  \texttt{(PROP ( )}
  \texttt{LOCAL (temp \_p p0; temp \_x (Vint (Int\_repr (Z\_of\_nat x))); temp \_value v)}
  \texttt{SEP (treebox\_rep t0 p0))}
  \texttt{LOOP}
  \texttt{POSTCONDITION}
\end{verbatim}

Since the loop body does not have a \texttt{break} or \texttt{continue} command, we can verify this loop with the single-invariant version of \texttt{forward\_loop} (lines 4 - 7).

Lines 8 - 12 proves that the precondition implies the loop invariant. \texttt{Exists} and \texttt{entailer!} are VST-Floyd’s tactics for proving assertion entailments. Here, “\texttt{Exists p0 t0 (fun t: tree val ⇒ t)}” instantiates the existentially quantified variables on the right side. And ‘\texttt{entailer!}” simplifies the leftover separation logic entailment. Afterwards, we apply the following domain-specific theorem:

\begin{verbatim}
\texttt{emp\_partial\_treebox\_rep\_H}
  : \forall p : \texttt{val}, \texttt{emp} \vdash \texttt{partial\_treebox\_rep} (fun t : tree val ⇒ t) p p
\end{verbatim}

The proof for loop body is longer but it mainly follows the C program’s control flow.
In the end of first branch, we need to prove the following entailment (before line 33 in the Coq proof):

\[
p0 : \text{val} \\
x : \text{nat} \\
v : \text{val} \\
H : \text{Int.min\_signed} \leq \text{Z.of\_nat} x \leq \text{Int.max\_signed} \\
H0 : \text{is\_pointer\_or\_null} v \\
t0 : \text{tree val} \\
p : \text{val} \\
P : \text{tree val} \rightarrow \text{tree val} \\
\text{P (insert x v N)} = \text{insert x v t0} \\
q : \text{val} \\
\text{data\_at Tsh t\_struct\_tree} \\
\text{(Vint (Int.repr (Z.of\_nat x)), (v, (Vint (Int.repr 0), Vint (Int.repr 0)))) q} * \\
\text{data\_at Tsh (tptr t\_struct\_tree) q p} * \text{partial\_treebox\_rep P p0 p} \\
\vdash \text{treebox\_rep (insert x v t0) p0}
\]

We first sep-apply (treebox\_rep\_leaf x q p v) which will simplify the left side of the entailment (sep-apply is another VST-Floyd’s tactic for proving entailments). Here is the theorem statement of treebox\_rep\_leaf:

\[
\text{treebox\_rep\_leaf} \\
: \forall (x : \text{nat}) (p b v : \text{val}), \\
\text{is\_pointer\_or\_null v} \rightarrow \\
\text{Int.min\_signed} \leq \text{Z.of\_nat} x \leq \text{Int.max\_signed} \rightarrow \\
\text{data\_at Tsh t\_struct\_tree} (\text{Vint (Int.repr (Z.of\_nat x)), (v, (nullval, nullval))}) p * \\
\text{data\_at Tsh (tptr t\_struct\_tree) p b} \vdash \text{treebox\_rep (T N x v N) b}
\]
Here is the proof goal afterwards:

\[
p0 : \text{val} \\
x : \text{nat} \\
v : \text{val} \\
H : \text{\texttt{Int.min\_signed}} \leq \text{\texttt{Z.of\_nat}} \ x \ \leq \ \text{\texttt{Int.max\_signed}} \\
H0 : \text{\texttt{is\_pointer\_or\_null}} \ v \\
t0 : \text{tree val} \\
p : \text{val} \\
P : \text{tree val} \rightarrow \text{tree val} \\
H1 : P (\text{\texttt{insert}} \ x \ v \ N) = \text{\texttt{insert}} \ x \ v \ t0 \\
q : \text{val} \\
\]

\[
\text{\texttt{treebox\_rep}} \ (T \ N \ x \ v \ N) \ p \ \ast \ \text{\texttt{partial\_treebox\_rep}} \ P \ p0 \ p \ |- \\
\text{\texttt{treebox\_rep}} \ (\text{\texttt{insert}} \ x \ v \ t0) \ p0
\]

Now, “\texttt{rewrite \ltimes -H1}” and “\texttt{change (T N x v N) with (insert x v N)}” will leave us the following proof goal, which is obviously an instance of rule (7.2d) (rule (7.2d) is called \texttt{treebox\_rep\_partial\_treebox\_rep} in Coq).

\[
p0 : \text{val} \\
x : \text{nat} \\
v : \text{val} \\
H : \text{\texttt{Int.min\_signed}} \leq \text{\texttt{Z.of\_nat}} \ x \ \leq \ \text{\texttt{Int.max\_signed}} \\
H0 : \text{\texttt{is\_pointer\_or\_null}} \ v \\
t0 : \text{tree val} \\
p : \text{val} \\
P : \text{tree val} \rightarrow \text{tree val} \\
H1 : P (\text{\texttt{insert}} \ x \ v \ N) = \text{\texttt{insert}} \ x \ v \ t0 \\
q : \text{val} \\
\]

\[
\text{\texttt{treebox\_rep}} \ (\text{\texttt{insert}} \ x \ v \ N) \ p \ \ast \ \text{\texttt{partial\_treebox\_rep}} \ P \ p0 \ p \ |- \\
\text{\texttt{treebox\_rep}} \ (P (\text{\texttt{insert}} \ x \ v \ N)) \ p0
\]

Thus, “\texttt{apply treebox\_rep\_partial\_treebox\_rep}” can directly solve it.

The Coq proof techniques for other branches are similar.
Chapter 16

Related work

In this chapter, I will first generally compare automated verification and interactive verification (not only VST-Floyd) then compare VST-Floyd with other interactive verification tools.

16.1 Automated vs. interactive verification

Automated verification and interactive verification are two different kinds of verification approaches. Using automated verification tools, computers take the main responsibility for proving programs’ correctness. Specifically, humans write programs, humans write program specifications and computers test whether the programs obey their corresponding specifications. Using interactive verification tools, humans are responsible for building correctness proofs. In other words, humans write programs and specifications, humans prove that the programs obey their corresponding specifications and computers check whether every step in these proofs follows basic logical proof rules.

Although these tools on both sides are so different from each other, their motivation is the same: testing whether what people believe to be true is actually true.
One on side, many programs are obviously safe (or correct)—for example, the functional correctness of summing up a linked list of integers, the memory safety of AES trivially implemented by arrays, etc. A programmer can easily read and understand such code, and check such correctness properties. A similar code review process can be done by a computer verifier—this is exactly what automated verification tools do. Modern automated verification tools are usually based on decision procedures like SMT solvers, BDDs, etc.

On the other side, some programs’ correctness is nontrivial—computer scientists prove their correctness instead of just believing it. Although constructing such correctness proofs is difficult, understanding these proofs is easy (either for a human reader or for a computer). A computer can check whether every single step in the proofs is valid—this is exactly what interactive verification tools do. Interactive verification tools are usually built on proof assistants (also called proof management tools) like Coq and Isabelle.

In comparison, automated verification represents a cheaper solution (since less human effort is required) for relatively easier problems and interactive verification represents an more expansive solution for relatively harder problems.

Traditionally, due to the undecidability theorem, most automated verification researchers try to find out those properties which are (1) weak enough so that computers can efficiently decide (in most cases) whether a program satisfies them or not, and (2) strong enough to ensure critical safety criteria.

For example, the validity of classical propositional logic expressions is decidable. But after adding quantifiers, the validity of classical first order expressions is undecidable. CBMC [40] verifies C programs with assertions themselves defined in C (as C functions with boolean-typed return values). That means, its assertion language
includes all boolean combination, arithmetic, and integer comparison but does not allow quantifiers.

For automated verification tools, separation logic is a technique to avoid using quantifiers when describing memory aliasing. For example, to express the following assertion without separation logic

\[
\text{listrep}(p, l_1) \ast \text{listrep}(p, l_2),
\]

one has to quantify over the address footprints of two linked lists.

For most fragments of first-order logic, validity is undecidable. But many widely used quantifier-free domain-specific separation logics are decidable.

Smallfoot [11] is an automated verification tool with a decision procedure for linked-list and binary tree shape analysis (in separation logic). Grasshopper [47] reduces shape analysis of linked lists and trees (in separation logic) to graph reachability related problems and designs a decision procedure for that.

Recently, some verification tools blur the boundary between the automation side and the interactive side. Hip/Sleek [20] uses a heuristic strategy to handle inductively defined separation logic predicates. Its algorithm is incomplete but can practically verify many programs about linked lists and binary trees. Hip/Sleek’s application is not even limited to shape analysis—it can verify some separation logic specification about functional correctness.

Dafny [41] is a non-separation-logic-based program verification tool in which users are responsible for offering some proof hints. Specifically, users must write assertions in programs. These assertions can guide the verifier to prove the program correct. When this is still not enough, users can write auxiliary lemmas, which Dafny automatically proves, then automatically applies in the program verification.

\footnote{In principle, loops or recursive functions can implement quantifiers in C. But CBMC does not have real loops or recursions. It only iterates loops or executes recursive calls for a bounded number (set by users) of times.}
Such combination of incomplete (but practical) decision procedures and users’ hints has made these verification tools not very different from interactive ones. In comparison, VST’s users use tactics to guide program correctness verification.

Besides the extent of automation, some interactive verification tools differ from automated ones in the tools’ own correctness guarantee. A C program’s correctness proof in VST is sound w.r.t. CompCert’s C semantics. This soundness is ensured by Coq’s tiny logic kernel—CIC. Iris and Charge! are two other separation logic verification tools. All their program correctness proofs can also be reduced to Coq’s logic kernel. CakeML is a verification tool built on Isabelle/HOL. Its own correctness is ensured by higher order logic (HOL). Research scientists can also choose other proof management software like Isabelle/ZF (based on set theory), ACL2, Agda and etc.

In contrast, there is no automated verification tool which is foundationally proved sound. Having these tools themselves verified (i.e. proving their functional correctness) is nontrivial.

The first problem is how to formally state “the correctness of an automated verification tool”. In principle, this correctness property can only be expressed in a metalanguage. For now, that metalanguage can only be a formal proof assistant like Coq and Isabella. Since the verifiers’ correctness are not formally stated, it is unclear whether the specification that a verifier proves will be preserved by a compiler. For example, Hip/Sleek verifies programs of a domain-specific language. This language has a C-like syntax and a Java-like memory model. Specifically, consider the following C struct definition and separation logic assertion:

\[\text{In Hip/Sleek, users do not use assertions as hints but add rewrite rules as hints. These rewrite rules are logical equivalence between two separation logic assertions—users are responsible for their correctness.}\]

\[\text{Interactive verification tools are not necessarily sound w.r.t. a programming language’s operational semantics. Computer scientists may build an interactive tool which is only proved sound w.r.t. a program logic.}\]
struct IntPair { int fst; int snd; };

\( (p_1 \xrightarrow{\text{IntPair}} 0,0 \ast p_2 \xrightarrow{\text{IntPair}} 0,0) \land (q \xrightarrow{\text{IntPair}} 0,0 \ast T) \)

In Hip/Sleek, this assertion implies \( p_1 = q \lor p_2 = q \) because in a Java-like memory model, \texttt{IntPair} represents an object and one object’s memory cannot partially overlap with another one’s. In other words, two objects are either identical (and occupy the same piece of memory) or occupy two disjoint pieces of memory. But this is not true in C memory model, it is also possible that \( p_1 = q - 4 \) and \( p_2 = q + 4 \) (if an integer is 4 bytes).

Besides the difficulty of formally stating verifier’s correctness, here is another problem of verifying automated verification tools themselves: when an automated verification tool grows or gets improved, its correctness cannot be simply reduced to a small kernel. Usually, an automated verification tool is based on a SAT solver. Thus, a tool’s correctness is dependent on the SAT solver’s correct implementation. Whenever adding a new theory solver to a verification tool, the whole tool’s correctness becomes dependent on that new solver as well.

### 16.2 Deep embedding vs. shallow embedding

Deep embedding and shallow embedding are two different approaches to formalizing programs, assertions, entailments and Hoare triples in a proof assistant like Coq.

CompCert formalizes the C language (and all intermediate languages in compilation) by deep embedding. In other words, a C command is defined as a syntax tree in Coq. Here is a segment of CompCert Clight’s formalization:\(^4\)

---

\(^4\)Clight is one of the earliest stages in compiling. Verifiable C and VST-Floyd actually verify Clight code generated by CompCert Clightgen from C code.
**Inductive** statement : Type :=

| Sskip : statement (* do nothing *)
| Sassign : expr → expr → statement (* assignment [lvalue = rvalue] *)
| Sset : ident → expr → statement (* assignment [tempvar = rvalue] *)
| Ssequence : statement → statement → statement (* sequence *)
| ...

In addition, the meaning of a C command—the operational semantics—is defined as an inductive relation in Coq.

FSCQ [19] formalizes the programming language for building a file system by shallowing embedding. In other words, every program is defined as a function from program states to program states (their programs are always deterministic). Its operational semantics does not need to be defined separately—that function from program states to program states is the program’s semantic meaning.

In comparison, shallow embedding is more flexible for adding new language features in the future. Adding a new language pattern like \( x++ \); is almost no cost for shallow embedding; no existing proofs (of properties of the language) will break. For deep embedding, this means adding one more inductive constructor in the definition of syntax tree. That may break legacy proofs—that is, any proof by induction (or by case analysis) over the syntax or over the operational semantics.

As a trade-off, we cannot define concepts like “the size of a program” or “the constant propagation transformation” on shallowly embedded languages. For example, the following programs show an example of constant propagation.

---

5When Chen et. al turned to verify concurrent file systems later, they made their language deeply embedded.

6Currently, “\( x++ \);” is not a Clight command. It will be rewritten into “\( x = x + 1; \)” when CompCert translates C programs to Clight programs.
\[\begin{align*}
\text{Before constant propagation}\\
x &= 5; \\
x &= x + 5; \\
y &= y - x; \\
x &= 0; \\
\end{align*}\]

\[\begin{align*}
\text{After constant propagation}\\
y &= y - 10; \\
x &= 0; \\
\end{align*}\]

They are two different programs from deep embedding’s point of view since their syntax trees are different. But they represent the same mapping from program states to program states. Thus, they are indistinguishable from shallow embedding’s point of view. So, it is impossible to define a Coq function which transform the first program into the second one under the shallow-embedding formalization.

Like programming languages, an assertion language can be shallowly embedded or deeply embedded. Verifiable C chooses shallow embedding. But the canonical form of VST-Floyd is semideep. Specifically, a canonical assertion is composed of a PROP list, a LOCAL list and a SEP list. Its meaning is defined by iterated conjunction and iterated separating conjunction. The contents in PROP and SEP clauses are shallowly embedded. The contents in LOCAL clauses are mostly deeply embedded, e.g. “temp x (a + 1)” means \([x] = a + 1\). Here, only the expression \(a + 1\) is shallowly embedded. This combination of shallow embedding and deep embedding enables us to build efficient (computational) strongest-postcondition generators.

Entailments and Hoare triples can also be deeply embedded or shallowly embedded. Verifiable C also chooses shallow embedding. As described in section \[10.1.2\] an entailment between two separation logic assertions is formalized as a Coq proposition “\(\text{derives } P \ P\)”: for any stack \(s\) and heap \(h\), if \((s, h)\) satisfies \(P\) then it also
satisfies Q. The Hoare triple $\Delta \vdash \{P\} \ c \ \{Q\}$ is formalized as a Coq proposition: “$\text{semax} \ \Delta \ P \ c \ (Q_1, Q_2, Q_3, Q_4)$” That is, for any program state $(s_1, h_1)$, if it satisfies $P$, then (1) it is safe to execute $c$ from it; (2) if such execution terminate with a program state $(s_2, h_2)$ normally, by break, by continue or by return, then $(s_2, h_2)$ should satisfy $Q_1$, $Q_2$, $Q_3$ or $Q_4$ respectively.

Section 14.5 shows the possibility of formalizing Hoare triples by deep embedding. Specifically, $\Delta \vdash_{\text{min}} \{P\} \ c \ \{Q\}$ is a proof tree of some specific proof theory. As a comparison, we get the power of doing induction over the proof trees.

16.3 Other interactive verification tools

VST is not the only interactive program verification research project. I discuss other related research projects and verification tools in this section.

CertiKOS verifies an operating system implemented in C and assembly. They mainly use the CompCert’s formalization of C semantics and assembly semantics. In other words, the programming languages are deep embedded. CertiKOS does not use a program logic in verification. Program correctness is directly proved by analyzing the operational semantics. In general, it is inconvenient. But for the specific program they verify, the operating system, it turns out to be a not too big problem. The most important contribution of CertiKOS is the verified abstraction layers. Specifically, all C (or assembly) functions implemented in their operating system can be divided into several layers, one based on another. These C (or assembly) functions are proved correct with respect to their specifications. The specification at every single layer abstracts away the implementation details of all lower-layer programs. The verified abstraction layers enable CertiKOS to organize the operating system’s correctness proof in an elegant way.
FSCQ uses separation logic to verify a file system. FSCQ formalizes the programming language by shallow embedding. In other words, their programs are always deterministic and every program is defined as a function from program states to program states. When they turned to verify concurrent file systems later, they made their programming language and operational semantics deeply embedded. They develop a shallow embedded separation Hoare logic to handle system crashes and system resumption.

Bedrock is a program logic and tool for reasoning about low-level (idealized assembly language) programs. Its assertion language uses lambda calculus and directly refers to stack and heap. To express the following canonical assertion in VST-Floyd,

\[ \text{PROP}() \text{LOCAL}(\text{temp} \times p) \text{SEP}(p \mapsto 0) \]

Bedrock users directly write: \( \lambda s : \text{stack}. \lambda h : \text{heap}. s(x) = p \land h(p) = 0 \). VST-Floyd performs better than bedrock for two reasons. On one hand, the abstraction of separating conjunction enables concise representation of heap disjointness. \( \text{SEP}(p \mapsto 0; q \mapsto 0) \) implies the fact the \( p \neq q \). On the other hand, Coq’s tactic language does not work well in a subcontext. In the example above, the expression \( s(x) = p \land h(p) = 0 \) is in a subcontext with two extra variables \( s \) and \( h \). Pattern matching in Coq’s tactic language does not work very well in this scenario. Our separation logic predicates abstract away this lambda calculus. Our tactic library directly operates in the top level context and is thus easier to build.

Charge! is a program logic for Java, based on separation logic. It is not linked to a formally verified Java compiler, which means that one cannot consider its Java semantics fully “debugged”. Charge’s assertion language is mostly written in lifted separation logic while our canonical form mostly operates in the unlifted language (SEP part). Because of the isolation of C variables in our canonical form, our tactic
library only modifies one conjunct (in \textit{LOCAL} part or \textit{SEP} part) in a precondition to generate a postcondition. Until 2014, assertions in VST-Floyd were written in the lifted language instead of canonical forms. We found that proof rules were very complicated and tactics were very slow. For similar reasons, Charge! is less efficient than VST-Floyd (see section 11.6).

Besides isolating program variables and the usage of unlifted logic, the small scale domain specific reflection used in forward tactic implementation is another reason of VST-Floyd’s higher efficiency. No paper about Bedrock or Charge! published any performance numbers for separation-logic based program verification, as we have done [17]. But we have heard that some bedrock’s example are measured in tens of hours and tens of GB of RAM. In contrast, typical VST proofs need only minutes and less than one GB.

CakeML [28] is a verified compiler for ML. It is proved correct in Isabelle/HOL. It is now accompanied with a separation Hoare logic for verifying functional correctness. Their system is based on a Characteristic Formulae framework, which is similar to Hoare logic. The construction of Characteristic Formulae is actually a combination of Hoare rules and the definition of Hoare triple validity. The soundness of such construction corresponds to the soundness of Hoare rules.

Iris [35] is a general purpose modal concurrent separation logic, parameterized over programming languages and program semantics, embedded in (and proved sound in) Coq. Iris can be used both to prove a separation Hoare logic sound and to verify programs via a program logic.

Iris Proof Mode (IPM) [38] provides tactics and lemmas for separation-logic proofs in Iris. IPM implements a proof mode (IPM) inside another proof mode (Coq)\footnote{VST-Floyd can also be treated as a proof mode. But in that sense, Iris proof mode is more Coq like.}. Moreover, IPM has several features lacking in VST-Floyd: special handling of a “persistent” modality, and special tactics for the various modalities of the Iris logic.
Some of these modalities also appear in Verifiable C—they are used in proving our separation Hoare logic sound but are seldom used in verifying programs.

For program logic’s soundness proof, Verifiable C is currently proved sound by unfolding the semantic definitions of these modalities (and separation logic connectives). Iris accomplishes similar soundness proofs at the logic level. Due to IPM’s support, building such logic-level proofs is convenient.

For program verification, Iris chooses to expose their semantic definition of Hoare triple: \( \{P\} c \{Q\} \) means \( P \vdash \text{wp}_c (Q) \). This design choice has both pros and cons. On one hand, preconditions and left hand sides of entailments automatically share the same set of proof rules. \texttt{HOARE-EXISTS} in Verifiable C is just an instance of \texttt{exists}:\footnote{\texttt{exists}: If \( \Sigma; \Gamma; P \vdash Q \) and \( x \) does not freely occur in \( \Gamma \), \( Q \) or \( \vec{R} \) then \( \Sigma; \Gamma; \exists x. P \vdash Q \).}

\[
\text{HOARE-EXISTS:} \quad x \text{ does not freely occur in } \Gamma, Q \text{ or } \vec{R} \\
\text{then } \Sigma; \Gamma; \Delta \vdash \{\exists x. P\} c \{Q, [\vec{R}]\}.
\]

\[
\exists\text{-elim:} \quad \text{If } \Sigma; x; \Gamma; P \vdash Q \text{ and } x \text{ does not freely occur in } \Gamma \text{ or } Q \\
\text{then } \Sigma; \Gamma; \exists x. P \vdash Q
\]

On the other hand, \( \text{wp}_c (Q) \) is the wrong abstraction level for Hoare logic proofs. A proof goal of form \( \Sigma; \Gamma; P \vdash \text{wp}_c (Q) \) sometimes confuses users—it is actually a Hoare triple but looks like an entailment. Another consequence of it is on the theory side: they would be unable to formalize their proof rules as a Hoare logic proof theory, as discussed in section 14.5.

Another difference between Iris’s and VST-Floyd’s design choices is how to implement forward verification tactics. Iris’s implementation is more tactic-based. In other words, they prove elementary proof rules sound (like Verifiable C’s proof rules in chapter 2) and build tactic programs by directly applying those rules. Our imple-
mentation is more theorem-based. We usually prove derived rules (as Coq theorems or lemmas) from elementary rules first and then build tactics on those derived rules. We choose this approach because C semantics has many subtleties. Using derived proof rules as an intermediate level makes the whole system convenient to maintain. After development a new feature, it is easy to tell whether a complicated legacy proof (which might be broken by the new feature) is debugged. In contrast, testing whether a complicated tactic is broken by a new feature is difficult.

As Krebbers et al. write, “[unlike IPM] all the tools that we are aware of are primarily focused on program verification.” Indeed, that is the focus of VST-Floyd. However, it should be possible to include some of our techniques into IPM and get the best of both worlds.
Chapter 17

Conclusion

This thesis presents a new technique \textit{magic-wand-as-frame} and a tool \textit{VST-Floyd} for separation-logic-based program verification formalized in Coq.

Magic-wand-as-frame is a general way of writing separation-logic assertions to describe partial data structures (or data structures with a hole). I have used \textit{magic-wand-as-frame} to verify C implementations of linked list operations, BST operations and a hash table library.

VST-Floyd is a C program verification tool which is foundationally sound (proved sound w.r.t. CompCert Clight’s operational semantics), flexible (easy to integrate with domain-specific theories, especially those domain-specific Coq theorems and tactics), convenient and efficient to use. VST-Floyd has been used to verify some classic algorithms and cryptograph primitives (see Figure 17.1).

The linked list append program’s correct proof presented in section 6.2 takes 186 lines of Coq code. It contains:

A. 17 lines of program specifications, in which 5 lines are the definition of \texttt{listrep(\_\_, \_\_)}.

B. 118 lines of auxiliary lemmas’ proof, in which 62 lines are lemmas for VST-Floyd’s automation including the fact that \texttt{listrep}(p, l) implies p’s being a
pointer or null, 31 lines are general lemmas for any list segment predicate and only 25 lines are specifically separation logic entailments about \( \text{lseg}(\_ , \_ , \_ ) \).

Using magic-wand-as-frame enables us to reuse most of these proof script. In comparison to this number 25, proving the same program using \( \text{lseg}_\text{rec}(\_ , \_ , \_ ) \) and \( \text{nt}_\text{lseg}_\text{rec}(\_ , \_ , \_ ) \) needs 46 and 68 lines of proofs for similar separation logic entailments respectively.\(^1\)

C. 51 lines of VST-Floyd’s program correctness proof.

The BST insert program is verified by 353 lines of Coq code, in which 67 lines are VST-Floyd’s program correctness proof (the proof in section 7.2) and 49 lines proves separation logic entailments for partial trees. I compare it with two other proof strategies. One still uses magic-wand-as-frame to define partial tree predicates but uses Coq’s inductive type to formalize partial trees (my proof uses functions between trees to represent partial trees):

```coq
Inductive partial_tree : Type :=
| H : partial_tree (* hole *)
| L : partial_tree \rightarrow key \rightarrow V \rightarrow tree V \rightarrow partial_tree (* hole in the left subtree *)
| R : tree V \rightarrow key \rightarrow V \rightarrow partial_tree \rightarrow partial_tree (* hole in the right subtree *).
```

Another uses traditional recursively defined partial tree predicates—this recursive predicate must take an inductively defined partial tree as its argument. These two alternative proofs take 111 lines and 137 lines for proving their partial-tree predicates’

\(^1\)Another 31 lines of proofs for general list segment properties are not counted here because of the following two reasons. (1) If I were not to prove the properties of \( \text{lseg}(\_ , \_ , \_ ) \) from general facts, it could be far less than 31+25 lines. (2) These 31 lines can be reused when proving other list segment predicates’ properties. For example, we may reimplement linked list append using pointer-to-pointer-to-list like the BST insert program in chapter 7. Then we can use these general lemmas to prove list box segment’s properties. (This alternative C implementation and its correctness proof is also formalized. The “loop+box” line in figure 17.1 shows the a detailed version of line count.) These general lemmas is also useful for a different C implementation of linked lists, e.g. using a different C struct.
<table>
<thead>
<tr>
<th>Linked list append</th>
<th>loop</th>
<th>lseg</th>
<th>15</th>
<th>$5^a+12$</th>
<th>$62^b+31^c+25$</th>
<th>51</th>
<th>75 (all)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>lseg_rec</td>
<td></td>
<td></td>
<td>$62^a+46$</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ntlseg_rec</td>
<td></td>
<td></td>
<td>$62^b+68$</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loop+box</td>
<td>lbseg</td>
<td>14</td>
<td></td>
<td>$62^b+31^c+29$</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recursion</td>
<td>N/A</td>
<td>11</td>
<td></td>
<td>$62^b+0$</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>BST insert</td>
<td>PTBrep+func</td>
<td>22</td>
<td>$17^d+12$</td>
<td>$180^b+28^d+49$</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PTBrep+ind</td>
<td></td>
<td></td>
<td>$180^a+111$</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PTBrep_rec</td>
<td></td>
<td></td>
<td>$180^b+137$</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST other</td>
<td></td>
<td>60</td>
<td>$17^d+63$</td>
<td>$180^a+28^d+98$</td>
<td>214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO Queue</td>
<td></td>
<td>82</td>
<td></td>
<td>530</td>
<td></td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>SHA</td>
<td></td>
<td>239</td>
<td></td>
<td>4956</td>
<td></td>
<td>525</td>
<td></td>
</tr>
<tr>
<td>HMAC</td>
<td></td>
<td>256</td>
<td></td>
<td>5432</td>
<td></td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>Salsa</td>
<td></td>
<td>280</td>
<td></td>
<td>5019</td>
<td></td>
<td>921</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td></td>
<td>240</td>
<td></td>
<td>1256</td>
<td></td>
<td>1209</td>
<td></td>
</tr>
</tbody>
</table>

a. These 5 lines are the definition listrep(_,_), shared by all linked-list-related program verification.
b. These 62 lines are lemmas for VST-Floyd’s automation, e.g. listrep(p,l) implies p’s being a pointer or null. They are shared by all linked-list-related program verification.
c. These 31 lines are general lemmas for any list segment predicate.
d. These 17 lines are the definition Trep(_,_). and TBrep(_,_).
e. These 180 lines are (1) recursive equations between Trep(_,_). and TBrep(_,_). (see Figure 7.4), and (2) lemmas for VST-Floyd’s automation.
f. These 28 lines are general lemmas for any list partial tree predicate like PTrep(_,_). and PTBrep(_,_).
g. Linked-list imperative FIFO queue, proved by Andrew Appel, 2012.[5 Chapter 28]
h. The OpenSSL implementation of the SHA-2 cryptographic hash algorithm, specialized to the 256-bit case. Proof done by Andrew Appel, 2013.[3]
i. The OpenSSL implementation of the HMAC cryptographic authentication algorithm; proof done by Lennart Beringer, 2014.[13]
j. Parts of TweetNaCl’s implementation of the stream cipher Salsa20[14]; partially done by Lennart Beringer, 2016.
k. The mbedTLS implementation of AES-256 symmetric-key encryption; proof done by Samuel Grueetter, 2017.

Figure 17.1: Programs verified by VST-Floyd
separation logic entailments respectively. (Other parts in the proofs are similar with mine. See Figure 17.1 for more details.)

In this thesis, Coq’s role is not only to show machine-checkable correctness of proofs. Coq also enables me to describe research results in a more formal way, e.g. the parameterized proof rules for partial tree predicates (rule (7.2c,7.2e)) and the main process of strongest postcondition generator for load commands (SEMAX-LOAD-CANON).
Bibliography


