上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：基于控制流分割的带注释程序验证

学生姓名：	周李韬
学生学号：	518030910407
专　　业：	计算机科学与技术 (IEEE)
指导教师：	曹钦翔
学 院 (系)：	电子信息与电气工程学院

基于控制流分割的带注释程序验证

摘　要

随着程序验证技术的发展，许多工具已经被开发出来用于程序正确性的验证。一类程序验证工具（如 VST）基于交互式定理证明工具（如 Coq），用户可以形式化地书写有关程序性质的形式化的定理和证明，并且通过定理证明器的检查，获得正确性的保证。另一类程序验证的方式是让程序员直接在代码中添加断言注释，由注释验证工具直接对程序进行验证。相比交互式定理验证工具，基于注释验证器的程序验证方法自动化程度更高，验证对程序开发过程产生的开销较低，但受限于注释语言的表达力，支持验证的性质和证明方法也不如基于交互式定理证明器的验证方法丰富。

本文通过扩展 VST 工具，开发了 VST-A 系统，使其能够在利用 VST 交互式定理证明的灵活性和表达力的同时，也能允许用户直接在程序中书写断言注释，达到半自动化地验证带断言注释的 C 程序的目的。具体而言，我们提出了一套通用算法，对 C 程序进行控制流分析，根据用户提供的断言注释进行验证目标的肢解。我们在 Coq 中，证明了肢解函数的可靠性，在证明过程中，我们发现 VST-A 系统的可靠性依赖于霍尔逻辑中的合取规则。我们基于 VST 的程序逻辑和语义模型，对函数调用、变量赋值等指令的“精确性”进行研究，确保肢解算法所依赖的霍尔逻辑合取规则成立，从而保证了整个系统端到端的可靠性。具体应用方面，本文开发的 VST-A 系统能被应用于各类 C 程序数据结构的验证中。

关键词：程序验证，分离逻辑，带注释程序，Coq

ASSERTION ANNOTATED PROGRAM VERIFICATION WITH CONTROL FLOW SPLITTING

ABSTRACT

With the development of program verification technology, many tools have been developed to verify the correctness of programs. Interactive verification tools (such as VST) are based on interactive theorem provers (such as Coq). Users can formally write formal theorems and proofs about program properties, and obtain correctness guarantees through the check of the theorem prover. Another approach of program verification is to use annotation verifiers. Annotation verifiers allow programmers to add assertion annotations directly into the code, and provide a verifier to validate the program directly. Compared with interactive verification tools, annotation verfiers have a higher degree of automation and bring lower burden to software engineers. However, due to the limited expressive power of the annotation language, the supported expression of program properties and proof methods are not as rich as those based on interactive theorem provers.

In this thesis, we extends VST to build VST-A. VST-A enjoys the flexibility and expressive power of interactive theorem provers, and also allows users to write assertion as comments directly into the program to achieve semi-automatic verification. To be specific, we propose a general algorithm to analyze the control flow of C programs and split the verification of a program into the verification of several paths separated by assertion comments. We prove the soundness of the split function in Coq. During the proof, we find that the soundness of VST-A relies on the conjunction rule in the Hoare logic. By looking into the semantic models of VST, we study the "preciseness" of function call, variable assignment and other instructions to ensure the conjunction rules holds in the Hoare logic on which our VST-A depends, thus enabling foundational soundness of the whole system. The VST-A system can be used to verify various programs manipulating data structures in C programs.

Key words: Program Verification, Separation Logic, Annotated Programs, Coq

Contents

Chapter 1 Introduction

In the past 15 years, researchers have built series of tools for program verification. These tools are used in different ways and have their own advantages.

Tools like CakeML[1], Iris[2, 3], VST[4-6] (interactive program verification tools) are built in interactive theorem provers like Coq[7] or Isabelle[8]. Users are required to write formal program correctness proofs in the same theorem prover.

One significant advantage of interactive tools is that these tools themselves are *foundationally sound* (i.e. have a formal proof w.r.t. the language's operational semantics in the proof assistant). This is especially meaningful for verifying real world programming languages and for verifying higher-order properties like functional correctness. On one hand, real world programming languages are complicated. For example, it is very subtle what C programs may cause undefined behavior. On the other hand, an advanced program logic for higher-order properties usually has a nontrivial soundness proof. For example, VST and Iris use step-indexed semantics for interpreting impredicative assertion languages whose soundness proof is complicated.

Interactive program verification tools also benefit from *the rich language* of Coq (and Isabelle, etc.) for writing formal definitions and proofs. These tools can easily have higher-order functions and predicates in their assertion languages. It is convenient to introduce additional logical connectives and unrestricted user-defined predicates. Additionally, tactic proof languages in theorem provers are very powerful for users to describe different proof strategies like proof-by-induction and proof-by-contradiction.

Another common fashion for program verification is to require programmers to write annotations in the source code. With sufficient assertion annotations, tools like Dafny[9], Hip/Sleek[10], VeriFast[11], Viper[12], and Frama-C[13] are able to verify program correctness automatically. By restricting the assertion languages, tools like CBMC[14], F-Soft[15, 16] and Infer[17] can reduce the annotation overhead for programmers while preserving automation.

Comparing with writing tactical proofs in a theorem prover, *writing annotations is a much more straightforward way of demonstrating that a program is correct.* For example, Figure 1–1 shows an implementation of in-place linked list reverse and its functional correctness proof. ① We use With to introduce a universally quantified logical variable l to represent the contents of the linked list. In annotated programs, the assertions clearly state what criteria the program state should have at every point. By contrast, key insights of program correctness easily get mixed with structural tactics as lengthy proof scripts in interactive verification tools.

In this thesis, we show that the benefits of interactive tools and annotation verifiers can be largely combined. We build VST-A, a foundationally sound verification tool, on top of VST. VST-A enjoys rich assertion languages and flexible proof strategies. Meanwhile, it allows users to write readable assertion annotations directly as comments like Figure 1–1.

The verification framework we propose for VST-A is illustrated in Figure 1–2. (1) Users should

① This is a separation logic[18] proof. In other words, we use an assertion of form "$P * Q$" to say that the memory can be split into two disjoint pieces of which one satisfies P and the other satisfies Q. ll($[\![\mathsf{p}]\!], l$) is a separation logic predicate asserting on the location referenced by variable p stores a linked list of l.

```
1  struct list {unsigned head; struct list *tail;};
2  struct list *reverse (struct list *p) {
3        /* With    l,
4           Require   ll ([[p]], l)
5           Ensure    ll ([[ret]], rev(l))  */
6        struct list *w, *t, *v;
7        w = NULL; v = p;
8        while (v) {
9        /* Assert   ∃ l1 c x l2'. l = rev(l1)  x  l2' ∧  [[v]] ↦ (x, c)  *  ll ([[w]], l1)  *  ll (c, l2')  */
10            t = v->tail; v->tail = w; w = v; v = t;
11       }
12       return w;
13 }
14
```

Figure 1–1 Annotations for verifying linked-list reverse

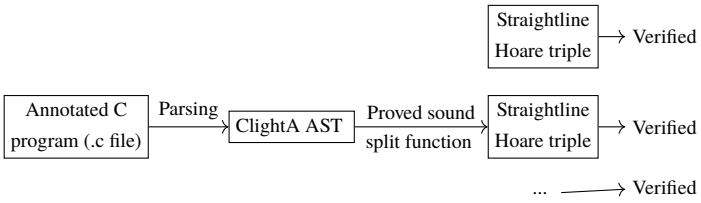

Figure 1–2 Verification workflow in VST-A

first provide a C program with sufficient assertion annotations. Our frontend parser will convert the source code into ClightA, a formal language for assertion annotated C programs. In a typical forward verifier that supports existentially quantified assertions, existential variables can be extracted to the proof context, so later proofs can refer to them. VST-A also supports such flexible usage of existential variables. We achieve this by defining a new dedicated syntax in ClightA that makes existential variables and their binding scopes explicit. (2) Next, a *split function* takes a ClightA program and its pre-/post-conditions as input, and returns a set of straightline Hoare triples, which only consists of a sequence of primary statements or expression checking conditions. For example, Figure 1–3 shows the split result of the reverse function in Figure 1–1. Four triples are returned as verification goals. The control flow graph in Figure 1–4 shows that all control flow paths separated by assertion annotations in the source program are captured in the split result. (3) Finally, users are left to prove each straightline Hoare triple in the split result. The soundness theorem of VST-A will ensure correctness of the original program if all paths have been verified.

In addition to foundational soundness, rich assertion languages, flexible proof strategies, and intuitive assertion based verification, VST-A also has the advantage of saving proof recompilation time. When a verified program is slightly updated, its correctness proof needs some corresponding updates. In interactive verifiers like VST, users need to run all tactical proof scripts in old versions, even though they only need to update a small portion of that. Now, only the changed part of the program and proof needs to be recompiled since other parts of reduction result is unchanged.

Our main problem in this new path-splitting approach is to establish its foundational sound-

(1) With l,
/* $\mathrm{ll}([\![\mathsf{p}]\!], l)$ */
`struct list *w, *t, *v;`
`w = NULL; v = p;`
`assume v;`
/* $\exists\ l_1\ c\ x\ l_2'.\ l = \mathrm{rev}(l_1)\ x\ l_2' \wedge$
$[\![\mathsf{v}]\!] \mapsto (x, c)\ *\ \mathrm{ll}([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}(c, l_2')$ */

(2) With l,
/* $\mathrm{ll}([\![\mathsf{p}]\!], l)$ */
`struct list *w, *t, *v;`
`w = NULL; v = p;`
`assume !v;` `return w;`
/* $\mathrm{ll}([\![\mathsf{ret}]\!], \mathrm{rev}(l))$ */

(3) With $l\ l_1\ c\ x\ l_2'$,
/* $l = \mathrm{rev}(l_1)\ x\ l_2' \wedge\ [\![\mathsf{v}]\!] \mapsto (x, c)\ *$
$\mathrm{ll}([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}(c, l_2')$ */
`t = v->tail; v->tail = w;`
`w = v; v = t;`
`assume v;`
/* $\exists\ l_1\ c\ x\ l_2'.\ l = \mathrm{rev}(l_1)\ x\ l_2' \wedge$
$[\![\mathsf{v}]\!] \mapsto (x, c)\ *\ \mathrm{ll}([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}(c, l_2')$ */

(4) With $l\ l_1\ c\ x\ l_2'$,
/* $l = \mathrm{rev}(l_1)\ x\ l_2' \wedge\ [\![\mathsf{v}]\!] \mapsto (x, c)\ *$
$\mathrm{ll}([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}(c, l_2')$ */
`t = v->tail; v->tail = w;`
`w = v; v = t;`
`assume !v;` `return w;`
/* $\mathrm{ll}([\![\mathsf{ret}]\!], \mathrm{rev}(l))$ */

Figure 1–3 Split results for verifying linked-list reverse

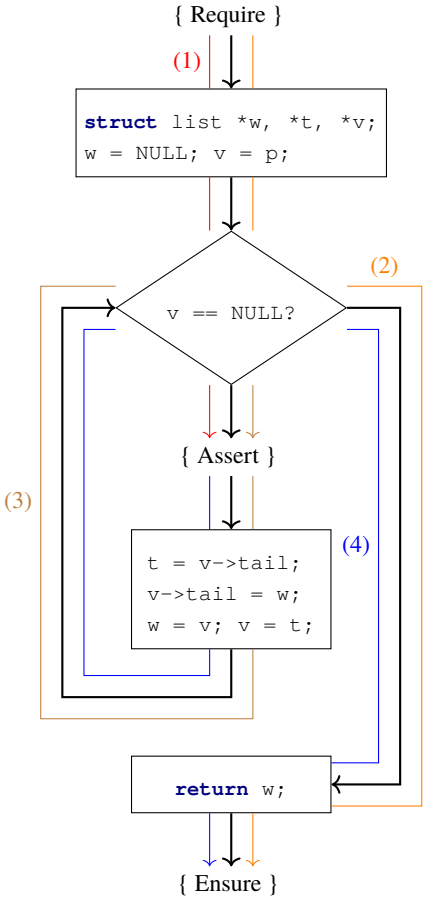

Figure 1–4 Control flow graph of linked-list reverse

ness. We implement the split algorithm mentioned above as a Coq function, and formally prove its soundness. Quite interestingly, we find that it seems inevitable to use the conjunction rule in this soundness proof. For example, say we would like to prove

```
/* P */ c1; if (b) { c2 /* Q */ c3 } else { c4 /* Q' */ c5 } /* R */
```

given that the following split results hold:

```
/* P */ c1; assume b; c2 /* Q */
/* Q */ c3  /* R */
/* P */ c1; assume !b; c4 /* Q' */
/* Q' */ c5 /* R */
```

We need a middle condition S such that the following two triples are provable

```
/* P */ c1 /* S */    /* S */ if (b) { c2 /* Q */ c3 } else { c4 /* Q' */ c5 } /* R */
```

We can construct the middle condition as follows:

```
S ≜ wp (assume b; c2, Q)  ∧ wp (assume !b; c4, Q')
```

where $\mathsf{wp}(c, Q)$ is the weakest precondition of c for postcondition Q. The second triple for if-branching is provable from the definition of weakest precondition and the triples about c_3 and c_5 in the premise. To prove the other half, triples about c_1 in the premise have the following two equivalent forms:

```
/* P */ c1  /* wp (assume b; c2, Q) */      /* P */ c1  /* wp (assume !b; c4, Q') */
```

Even for this simple example, we need to apply the conjunction rule to the two formulas above in order to obtain /* P */ c_1 /* S */.

In theories of separation logics, the conjunction rule is closely related to the notion of *preciseness*. Traditionally, the preciseness restrictions are placed on assertion predicates. For example, in concurrent separation logic settings, it has been shown that the resource invariant describing the lock should be a precise predicate in order to make the logical system with the conjunction rule to be sound[19, 20]. In our task, we do not have built-in conjunction rule, but wants to derive the conjunction rule from a practical but complex logic like VST. We introduce a new notion of preciseness that is defined for the specification of an operation. We show that the new precise specification restriction is more general than the widely-used notion of precise predicates, for it allows richer properties to be specified.

This thesis targets at extending VST to VST-A, a system that can make use of the flexibility and expressibility of VST interactive theorem proving, while also allow users to write assertion annotations directly into the program, so as to achieve the purpose of semi-automatic verification of C program with assertion annotations. The contributions of this thesis are:

1. **A new framework of program verification**: We propose a foundationally sound framework to support verification for assertion annotated programs. Our approach combines the benefits of interactive provers with the readablity of annotated programs, and enables incremental program verification for incremental program development.
2. **A formal language for annotated programs**: We define ClightA, a formal language of annotated C programs. This language not only defines C statements and assertions, but also deals with logical variables in the verification context.

3. **A control flow based verification splitting algorithm**: The foundational soundness of VST-A is based on a split function, which is implemented in Coq and proved sound w.r.t. VST semantics. We believe this splitting algorithm and its soundness proof is general and can be applied to other Hoare-style verification tools as well.
4. **A restudy of the conjunction rule and preciseness**: We restudy the conjunction rule in the setting of VST-A, and propose a new notion of preciseness, which is more general than the traditional notion of precise predicates, to derive the conjunction rule.

Chapter 2 Background and overview

In this chapter, we briefly introduce some background knowledge required by the rest of the thesis, and then we present an overview of the main components of our VST-A framework.

2.1 Background

2.1.1 Coq: interactive theorem prover

Coq[21] is an interactive theorem proving tool. Users can write machine-checked formal theorems and proofs rigorously, and obtain the guarantee of correctness through the theorem prover's checking. Machine checking gives users much more confidence that the proofs are correct compared to human-generated and -checked proofs. As a result, Coq has been successfully applied in many formal verification tasks, such as verifying the proof of the four color theorem. It has also served as the trusting base for the CompCert verified C compiler[22]. The thesis will focus on Coq's application as an interactive program verification tool, and base our development upon Verified Software Toochain (VST), a framework for verifying C program's functional correctness built in Coq.

In Coq, users generate proofs by entering a series of tactics that constitute steps in the proof. Besides the built-in tactics that Coq natively supports, users can also write their own tactics, in a programming approach with the Ltac2 tactic language that Coq integrates. Using Ltac2, large proofs can be simplified and automated.

The Coq kernel, a small part of Coq, does the final verification that the tactic-generated proof is valid, so the trusting base of Coq is minimal enough to be accepted as true by users. Usually the tactic-generated proof is indeed correct, but delegating proof verification to the kernel means that even if a tactic is buggy, it won't be able to introduce an incorrect proof into the system.

Finally, Coq also supports extraction of verified programs to programming languages such as OCaml and Haskell. This provides a way of executing Coq code efficiently and supplementing Coq functions with flexible I/O capabilities.

2.1.2 Hoare logic and separation logics

Hoare logic[23] is a theoretical method to describe and prove the correctness of programs. We can use Hoare logic to state what is the correct behavior of a program, and apply Hoare logic rules to prove that the program is correct.

the judgement of a Hoare logic is called a *Hoare triple*. A Hoare triple is a claim about the state before and after executing a command. A standard notation is $\{P\}c\{Q\}$, meaning:

- If command c begins execution in a state satisfying assertion P,
- and if c eventually terminates in some final state,
- then that final state will satisfy the assertion Q.

Assertion P is called the precondition of the triple, and Q is the postcondition. In this thesis, since we focus on verifying a program with assertion annotations, we will also use the symbol of C

comments such as $/*P*/$ to represent a condition that the program state at certain execution points should satisfy.

The goal of Hoare logic is to provide a compositional method for proving the validity of specific Hoare triples, i.e. to enable a Hoare triple of a program to be derived from the Hoare triples of its components. Each Hoare logic rule corresponds to different syntactic forms in the programming language to be verified. "Strucutral rules" are to be applied to achieve compositional reasoning. For example, if users want to verify a sequencing of two commands, the following Hoare logic rule can be applied.

$$\textsc{Hoare-Seq}\ \frac{\{P\}c_1\{R\} \qquad \{R\}c_2\{Q\}}{\{P\}c_1;c_2\{Q\}}$$

Usually, the assertion language of a Hoare logic at least contains propositional connectives: conjunction, implication etc. However, traditional Hoare logic suffers from the problem of aliasing when dealing with complex programs that manipulate the heaps. In programs with dynamic pointers or complicated data structures, assertions often need to talk about whether two pointers point to different locations in the heap. Otherwise, the program may not behave as expected. This is more formally stated as the *frame* problem: assertions need to describe the fact that when we manipulate one piece of the memory, other pieces of the memory are not affected.

Separation logic[18] addresses this problem by extending Hoare logic's assertion language with two extra connectives - separating conjunction ($P*Q$, or "star") and separating implication ($P \mathrel{-\!\!*} Q$, or "magic wand").

The separating conjunction $P*Q$ in assertions represents the existence of two disjoint portions of the state, one that satisfies P and one that satisfies Q. Specifically, a program model

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

Here, $m_1 \oplus m_2$ represents the disjoint union of two pieces of state/memory. Separating conjunction is commutative and associative. 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 $p \mapsto v * q \mapsto u$ 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 $p_1 \mapsto v_1 * p_2 \mapsto v_2 * \ldots * p_n \mapsto v_n$ 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:

$$m \vDash P \mathrel{-\!\!*} Q \text{ iff. for any } m_1 \text{ and } m_2, \text{ if } m \oplus m_1 = m_2 \text{ and } m_1 \vDash P \text{ then } m_2 \vDash Q$$

$$\textsc{Wand-Adjoint1}\ \frac{P \vdash Q \mathrel{-\!\!*} R}{P*Q \vdash R} \qquad \textsc{Wand-Adjoint2}\ \frac{P*Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

Besides basic Hoare logic proof rules like Hoare-Seq, separation Hoare logic enables one to verify a Hoare triple locally but use it globally, using the frame rule:

$$\textsc{Frame}\ \frac{\{P\}c\{Q\} \qquad \text{no program variables which freely occurs in } F \text{ are modified by } c}{\{P*F\}c\{Q*F\}}$$

2.1.3 Verified Software Toolchain (VST)

Verified Software Toolchain (VST)[24] is a mature tool for verifying the functional correctness of C programs. The tool, developed in Coq, is based on the extension of Hoare logic - separation logic. It allows users to write specifications for functions in the program and prove the correctness of the specifications by writing formal proofs in Coq. The main components of VST include

1. Verifiable C, the program logic for an abstract C language called Clight,
2. VST-Floyd[5], a proof automation system that applys Verifiable C to realistic program verification,
3. A machine-checked soundness proof that any program behavior proved in Verifiable C will be a subset of the defined behavior in the language's operational semantics.

In addition, the soundness proof of VST can be combined with the correctness proof of a verified C compiler - the CompCert project[22], so as to obtain the guarantee of end-to-end program correctness from C language to assembly language.

2.2 Preliminaries

We present a few key technical preliminaries of our framework in this section. Since the thesis is based on VST, most of the technical details will be presented from the perspective of VST.

2.2.1 Clight: abstract C language

In this thesis, we use a simplified version of CompCert Clight as an abstract C language. Figure 2–1 shows the syntax. $\mathsf{loop}\ (c_{\text{incr}})\ c$ is a general way to describe loops in the C language, which is equivalent to $\mathsf{for}\ (;;c_{\text{incr}})\ \{c\}$. Three kinds of loops in C language, namely for, while and do-while loops, can be expressed using this general loop statement. In return statements, return expressions are optional; thus we write $e?$ instead of e. Goto statement is not allowed — it is not supported by VST's program logic, and can be avoided by program transformation.

$$
\begin{array}{rrl}
\text{expression}: & e := & \cdots \\
\text{statement}: & c := & \mathsf{skip} \\
& \mid & c_1; c_2 \\
& \mid & \text{assignment statement} \\
& \mid & \mathsf{break} \mid \mathsf{continue} \mid \mathsf{return}\ e? \\
& \mid & \mathsf{if}\ (e)\ c_1\ \mathsf{else}\ c_2 \\
& \mid & \mathsf{loop}\ (c_2)\ c_1
\end{array}
$$

Figure 2–1 Clight: abstract C language

2.2.2 Forward symbolic execution

The verification backend of our work is based on a forward symbolic executor with two components: a symbolic assignment executor $\mathsf{ae}(\cdot,\cdot)$ and an assertion normalization program $\mathsf{norm}(\cdot)$. We write $\Sigma;\Gamma \vDash \mathsf{ae}(P,c) \Downarrow Q$ if Q is the result of symbolic execution, i.e., given logical variable

context Σ and proposition context Γ, $\mathsf{ae}(\cdot,\cdot)$ computes a strongest postcondition Q for precondition P and assignment statement c. For example, in the reverse program given in Figure 1–1, if

$$\begin{aligned}
&\Sigma := l,\\
&\Gamma := \emptyset,\\
&P := \exists\, l_1\ c\ x\ l_2'.\ l = \mathrm{rev}(l_1)\ x\ l_2' \wedge\ [\![\mathsf{v}]\!] \mapsto (x, c)\ *\ \mathrm{ll}\,([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}\,\left(c, l_2'\right),\\
&c := \mathsf{t = v\text{->}tail},
\end{aligned}$$

then $\Sigma;\Gamma \vDash \mathsf{ae}(P, c) \Downarrow Q$ for

$$Q := \exists\, l_1\ c\ x\ l_2'.\ l = \mathrm{rev}(l_1)\ x\ l_2' \wedge\ [\![\mathsf{t}]\!] = c\ \wedge\ [\![\mathsf{v}]\!] \mapsto (x, c)\ *\ \mathrm{ll}\,([\![\mathsf{w}]\!], l_1)\ *\ \mathrm{ll}\,\left(c, l_2'\right)$$

Assignment executor may fail, if the precondition P cannot guarantee c can run successfully, or P is not in a good form such that the symbolic executor can execute c.

Besides, an assertion normalization program turns an assertion into a *better* form. We write $\Sigma;\Gamma \vDash \mathsf{norm}(P) \Downarrow Q$ if the $\mathsf{norm}(\cdot)$ program terminates and returns Q when it tries to improve P. Here, P and Q should be equivalent with each other but Q is in a better form. For example, the following assertion I can serve as a loop invariant in our reverse program (Figure 1–1):

$$I := \exists\, a\ b\ l_1\ l_2.\ [\![\mathsf{w}]\!] = a\ \wedge\ [\![\mathsf{v}]\!] = b\ \wedge\ l = \mathrm{rev}(l_1)\ l_2\ \wedge\ \mathrm{ll}\,(a, l_1)\ *\ \mathrm{ll}\,(b, l_2)$$

So at the beginning of the loop body, the assertion $(I\ \wedge\ [\![\mathsf{v}]\!] = \mathsf{true})$ must hold. But this form may be not preferred by the symbolic executor and it becomes hard for next step. Assertion normalization program handles this situation. Here, $\Sigma;\Gamma \vDash \mathsf{norm}(I \wedge [\![b]\!] = \mathsf{true}) \Downarrow Q$ for

$$Q := \exists\, a\ b\ l_1\ l_2.\ [\![\mathsf{w}]\!] = a\ \wedge\ [\![\mathsf{v}]\!] = b\ \wedge\ b \neq \mathsf{NULL}\ \wedge\ l = \mathrm{rev}(l_1)\ l_2\ \wedge\ \mathrm{ll}\,(a, l_1)\ *\ \mathrm{ll}\,(b, l_2)\,.$$

VST provides an instance of forward symbolic executor inside.

2.2.3 Verifiable C

Figure 2–2 shows the structural proof rules that VST-A is based on. It is largely similar to the proof rules of the current VST release's Hoare logic. We put a few remarks on the common proof rules in this section.

The Hoare judgement in Verifiable C program logic extends the postcondition into four parts, to deal with control flow instructions such as break, continue and return?e. A judgement $\{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}$ can be interpreted as starting from a program state that satisfies P, after executing c, if the program exits normally, the program state will satisfy Q. Similarly, the program state should satisfies Q_{brk}, Q_{con}, and Q_{ret} when the program exits with a break status, a continue status, or a return status, respectively. We use $\left[\vec{Q}\right]$ as an abbreviation of the last three post conditions. We use Σ to denote the proof context of logical variables and Γ to denote the proof context of pure propositions.

The consequence rule Semax-Conseq says that under the proof context Σ and Γ, users can strengthen the precondition or weaken the postcondition by assertion entailment to prove a new Hoare judgement. In Verifiable C, assertion entailments are separation logic based, which will be explained in the main text where certain entailment rules are used.

$$\textsc{Semax-Conseq}\ \frac{\Sigma;\Gamma;P_1 \vDash P_2 \qquad \Sigma;\Gamma;R_2 \vDash R_1 \qquad \Sigma;\Gamma;\vec{R}'_2 \vDash \vec{R}'_1 \qquad \Sigma;\Gamma \vdash \{P_2\}\ c\ \left\{R_2, [\vec{R}'_2]\right\}}{\Sigma;\Gamma \vdash \{P_1\}\ c\ \left\{R_1, [\vec{R}'_1]\right\}}$$

$$\textsc{Semax-Skip}\ \frac{}{\Sigma;\Gamma \vdash \{P\}\ \mathsf{skip}\ \left\{P, [\vec{Q}]\right\}}$$

$$\textsc{Semax-Break}\ \frac{}{\Sigma;\Gamma \vdash \{Q_{\text{brk}}\}\ \mathsf{break}\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}}$$

$$\textsc{Semax-Continue}\ \frac{}{\Sigma;\Gamma \vdash \{Q_{\text{con}}\}\ \mathsf{continue}\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}}$$

$$\textsc{Semax-Return}\ \frac{}{\Sigma;\Gamma \vdash \{Q_{\text{ret}}\}\ \mathsf{return}\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}}$$

$$\textsc{Semax-Seq}\ \frac{\Sigma;\Gamma \vdash \{P\}\ c_1\ \left\{R, [\vec{Q}']\right\} \qquad \Sigma;\Gamma \vdash \{R\}\ c_2\ \left\{Q, [\vec{Q}']\right\}}{\Sigma;\Gamma \vdash \{P\}\ c_1; c_2\ \left\{Q, [\vec{Q}']\right\}}$$

$$\textsc{Semax-If}\ \frac{\Sigma;\Gamma \vdash \{P \wedge [\![b]\!] = \mathsf{true}\}\ c_1\ \left\{Q, [\vec{Q}']\right\} \qquad \Sigma;\Gamma \vdash \{P \wedge [\![b]\!] = \mathsf{false}\}\ c_2\ \left\{Q, [\vec{Q}']\right\}}{\Sigma;\Gamma \vdash \{P\}\ \mathsf{if}\ (b)\ c_1\ \mathsf{else}\ c_2\ \left\{Q, [\vec{Q}']\right\}}$$

$$\textsc{Semax-Loop}\ \frac{\Sigma;\Gamma \vdash \{I\}\ c\ \{I_{\text{con}}, [Q, I_{\text{con}}, Q_{\text{ret}}]\} \qquad \Sigma;\Gamma \vdash \{I_{\text{con}}\}\ c_{\text{incr}}\ \{I, [Q, \bot, Q_{\text{ret}}]\}}{\Sigma;\Gamma \vdash \{I\}\ \mathsf{loop}\ (c_{\text{incr}})\ c\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}}$$

Figure 2–2 Proof rules of C Hoare logic

$$\textsc{Semax-Set}\ \frac{}{\Sigma;\Gamma \vdash \{\triangleright\,(e \Downarrow \wedge P[e/\mathsf{x}])\}\ \mathsf{x} := e\ \left\{P, [\vec{\bot}]\right\}}$$

$$\textsc{Semax-Load}\ \frac{\pi \text{ is readable share}}{\Sigma;\Gamma \vdash \{\triangleright\,(\ \&e \Downarrow p \wedge (p \mapsto_\pi v * \mathsf{True}) \wedge P[v/\mathsf{x}])\}\ \mathsf{x} := e\ \left\{P, [\vec{\bot}]\right\}}$$

$$\textsc{Semax-Store}\ \frac{\pi \text{ is writable share}}{\Sigma;\Gamma \vdash \{\triangleright\,(\&e_1 \Downarrow p \wedge e_2 \Downarrow v' \wedge p \mapsto_\pi v * (p \mapsto_\pi v' \mathrel{-\!\!*} P))\}\ e_1 := e_2\ \left\{P, [\vec{\bot}]\right\}}$$

$$\textsc{Semax-Call}\ \frac{\Delta(f) = \phi \qquad \phi \text{ is a precise specification} \qquad \phi <: [\vec{A}]\{\lambda\vec{y}.P\}\,\{\lambda r.Q\}}{\Sigma;\Gamma \vdash \{\triangleright(\vec{e} \Downarrow \vec{b} \wedge \exists\vec{x} : \vec{A}.\ P\,\vec{b}\,\vec{x} * (Q\,\mathsf{a}\,\vec{x} \mathrel{-\!\!*} R))\}\ \mathsf{a} := f(\vec{e})\ \left\{R, [\vec{\bot}]\right\}}$$

Figure 2–3 Proof rules of primary C assignment commands

$$\textsc{Extract-Exists}\ \frac{\Sigma; x : A; \Gamma \vdash \{P\}\ c\ \left\{Q, [\vec{Q}']\right\}}{\Sigma; \Gamma \vdash \{\exists x : A.\ P\}\ c\ \left\{Q, [\vec{Q}']\right\}}$$

$$\textsc{Extract-Pure}\ \frac{\mathsf{pure}(P_{\mathsf{pure}}) \qquad \Sigma; \Gamma; P_{\mathsf{pure}} \vdash \{P\}\ c\ \left\{Q, [\vec{Q}']\right\}}{\Sigma; \Gamma \vdash \{P_{\mathsf{pure}} \wedge P\}\ c\ \left\{Q, [\vec{Q}']\right\}}$$

$$\textsc{Seq-Assoc}\ \frac{\Sigma; \Gamma \vdash \{P\}\ c_1; (c_2; c_3)\ \left\{Q, [\vec{Q}']\right\}}{\Sigma; \Gamma \vdash \{P\}\ (c_1; c_2); c_3\ \left\{Q, [\vec{Q}']\right\}}$$

$$\textsc{NoContinue}\ \frac{c \text{ contains no } \mathsf{continue} \qquad \Sigma; \Gamma \vdash \{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}}{\Sigma; \Gamma \vdash \{P\}\ c\ \left\{Q, [Q_{\mathrm{brk}}, Q'_{\mathrm{con}}, Q_{\mathrm{ret}}]\right\}}$$

$$\textsc{NoBreak}\ \frac{c \text{ contains no } \mathsf{break} \qquad \Sigma; \Gamma \vdash \{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}}{\Sigma; \Gamma \vdash \{P\}\ c\ \left\{Q, [Q'_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\right\}}$$

$$\textsc{NoReturn}\ \frac{c \text{ contains no } \mathsf{return} \qquad \Sigma; \Gamma \vdash \{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}}{\Sigma; \Gamma \vdash \{P\}\ c\ \left\{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q'_{\mathrm{ret}}]\right\}}$$

Figure 2–4 Derived rules from C Hoare logic

The Hoare triples for the basic control flow instructions are indicated by Semax-Skip, Semax-Break, Semax-Continue, and Semax-Return. For return statements, we only demonstrate a proof rule for statements without a return value. In our development, there is a similar proof rule with return value; we omit it here for simplicity.

To sequence two programs, similar to Hoare-Seq, in Verifiable C, Semax-Seq requires the a middle condition to be shared in the normal post condition of the first program and the pre-condtion of the second. Semax-If rule proves a if branching by asking for two Hoare judgements for the two branches, by introducing the expression checking condition into the pre-condition.

The loop structure makes use of the break and continue exit statuses, Semax-Loop rule has two invariants, loop invariant I and continue invariant I_{con}, that I is the invariant must hold before each iteration, and I_{con} is the invariant must hold at $\mathsf{continue}$ or before c_{incr} statement.

We leave the explanation for primary C statement rules in Figure 2–3 to Chapter 4. Rules Semax-Set and Semax-Load are explained in §4.3, Semax-Store is explained in §4.4, Semax-Call is explained in §4.5. The $\triangleright P$ predicate is a higher-order separation logic operator saying that instead of P we have a slightly weaker approximation to it. The theory of higher-order separation logic has been well studied, and the addition of higher-order separation logic does not affect the general implementation and correctness of VST-A, but increases the expressive power of the assertion and specification language. We will omit the explanation related to higher-order predicate operators in this thesis.

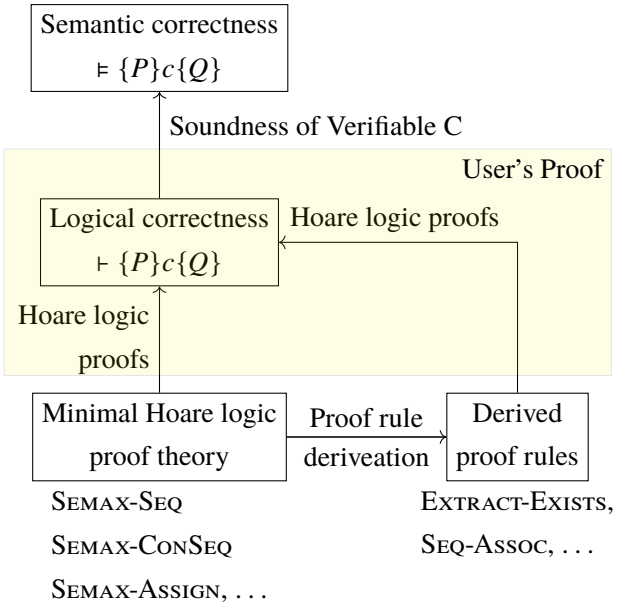

Figure 2–5 Framework of VST soundness

In addition to basic logical rules, useful rules in Figure 2–4 can be derived from rules in Figure 2–2 and Figure 2–3.

SEQ-ASSOC rule reassociates sequential compositions. Rules EXTRACT-EXISTS and EXTRACT-PURE rules introduce variables and propositions from precondition to context, respectively. Rules NOCONTINUE, NOBREAK, and NORETURN can rewrite a corresponding post-condition if the program contains no control flow instructions, which are used in our development for VST-A.

All of these rules are proved sound foundationally in VST w.r.t. CompCert Clight semantics.

2.3 Overview of the framework

In the following thesis, we begin presenting our new framework for verifying functional correctness of imperative programs with Hoare logic. The VST-A framework is based on the idea of splitting the verification of an assertion-annotated program into a set of control flow paths, so that users can achieve program correctness by verifying the correctness of each path separately. The benefit of this approach is that it makes the verification process modular and easy to automate. Besides, we believe that compared with writing proof scripts in interactive verification tools like VST (Verified Software Toolchain), writing assertions directly in imperative programs is more straightforward and easier to understand for software engineers. The framework is formalized as VST-A in Coq, with a mechanized proof of its soundness, so that VST-A can achieve the same foundational correctness guarantees as in VST.

Figure 2–5 and Figure 2–6 compare the common verification workflow in VST and our framework VST-A. VST provides a set of Hoare logic rules, which are proved sound with respect to the semantics of the language. To prove the correctness of a program c with respect to a pair of its pre-/post-condition specification P and Q, users need to write a proof script in the interactive theorem prover Coq, by applying the Hoare logic rules, and providing intermediate assertions (e.g. loop

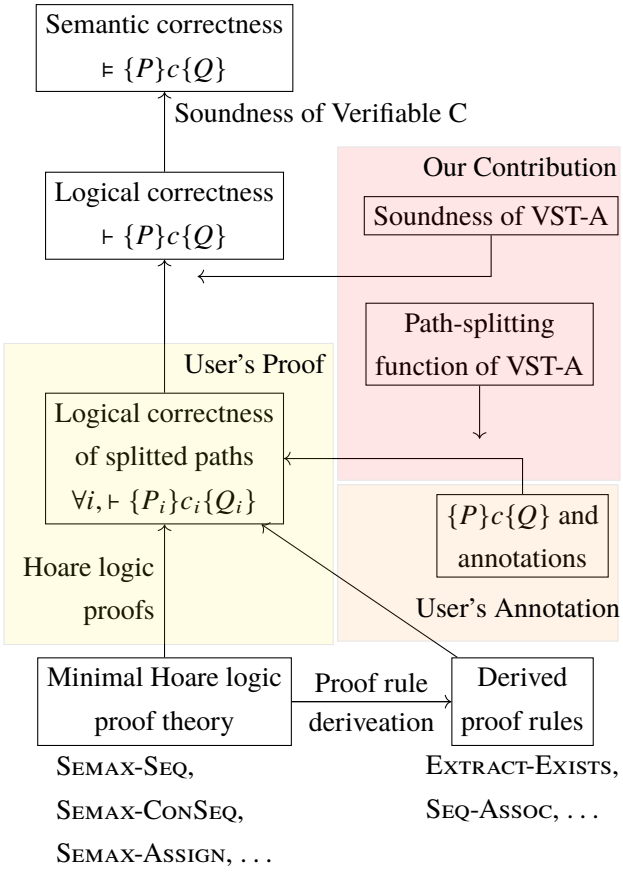

Figure 2–6 Overview of the VST-A framework

invariants) where necessary. The soundness of VST ensures end-to-end correctness of the proof.

In VST-A, we would like to allow users to verify a program by annotating programs with assertions, which constitutes the core wisdom of a program's verification. The verification workflow we advocate is depicted in Figure 2–6. Users first write assertions as comments in the source code of the programs, and then a path-splitting algorithm will automatically analyze the control flow between each two assertions and generate a set of paths $\{\{P_i\}c_i\{Q_i\}\}$ for users to verify independently. Since control flow information has been exploited, the commands of each path c_i are simply linear sequences of basic statements, which are left for users to verify. After all paths have been verified, the soundness theorem of VST-A will link them together, and together with the soundness theorem of VST, end-to-end verification is preserved.

The path-splitting function, and the mechanized proof of its soundness theorem are the two major contributions of VST-A against VST. We will introduce the path-splitting algorithm and the proof idea of soundness theorem in Section 3. Section 4 will discuss the "Conjunction rule", which we consider as a fundamental component of this framework.

2.4 Summary

In this chapter, we introduce the background of the thesis, including Coq, Hoare logic, separation logics, and Verified Software Toolchain (VST). Then we provide some preliminaries on VST

to support the introduction of our VST-A framework. Finally, we give an overview of the VST-A framework, including the main workflow, the path splitting procedure and the establishment of its soundness, mainly by comparing it with the structure of VST.

Chapter 3 Path split and its soundness

3.1 Clight-A: annotated C language

We define the annotation Clight-A syntax as an extension to the abstract C language of CompCert Clight in Figure 3–1. The Clight-A syntax provides extra constructors that allow users to insert assertions. The syntax of assertions is the same as defined in VST program logic, which we omit here.

The Clight-A syntax is implemented as an Inductive type in Coq. The complete Clight-A syntax C_c has two new constructors. Users can insert assertion P anywhere in the program through assert P constructor. Besides, if the assertion P has an existentially quantified logical variable x (in Coq type A), users may use the ExGiven $x : A, \{P(x)\}\ C_c$ constructor so that assertions that come after $P(x)$ in C_c can also refer to the logical variable x. With the two constructors, assertions in VST-A can match to the expressive power of corresponding proof scripts in VST.

To implement the path splitting procedure as a function in Coq, the complete Clight-A syntax C_c is parameterized by a simpler syntax C_s, which ignores the assertions contents and logical variables, and only specifies the shape of the assertion annotated program. To be specifc, the complete Clight-A syntax C_c is implemented as a dependepent type on a particular simple Clight-A syntax C_s in Coq as follows.

```
Inductive C_statement : S_statement -> Type := ...
```

Details for this design will be discussed in section 3.4

$$
\begin{array}{rrl}
\text{assertion}: & P := & \cdots \\
\text{expression}: & e := & \cdots \\
\text{primary statement}: & c_p := & e_1 := e_2 \\
& \mid & \mathsf{skip} \\
& \mid & a := f(\vec{b}) \\
\text{Clight statement}: & c := & c_p \\
& \mid & c_1; c_2 \\
& \mid & \mathsf{break} \mid \mathsf{continue} \\
& \mid & \mathsf{return}\ e? \\
& \mid & \mathsf{if}\ (e)\ c_1\ \mathsf{else}\ c_2 \\
& \mid & \mathsf{loop}\ (c_2)\ c_1 \\
\text{Simple Clight-A}: & C_s := & c \mid \mathsf{assert} \\
\text{Complete Clight-A}: & C_c := & c \mid \mathsf{assert}\ P \\
& \mid & \text{ExGiven}\ x : A, \{P(x)\}\ C_c
\end{array}
$$

Figure 3–1 syntax of Clight and Clight-A

$$
\begin{array}{rrcl}
\text{Basic statement}: & c_b & := & c_p \mid e \\
\text{Path atoms}: & p_{\_} & := & \vec{c_b} \\
\text{Path atoms with return}: & p_{\_}^{\text{ret}} & := & \vec{c_b}; ?e \\
\text{Simple pre-partial path}: & p_{s\dashv} & := & \vec{c_b}\dashv\{\} \\
\text{Complete pre-partial path}: & p_{\dashv} & := & \vec{c_b}\dashv\{P\} \\
\text{Simple post-partial path}: & p_{s\vdash} & := & \{\}\vdash\vec{c_b} \\
\text{Complete post-partial path}: & p_{\vdash} & := & \{P\}\vdash\vec{c_b} \\
 & & \mid & \forall(x:A).\ p_{\vdash} \\
\text{Simple post-return path}: & p_{s\vdash}^{\text{ret}} & := & \{\}\vdash\vec{c_b}; e? \\
\text{Complete post-return path}: & p_{\vdash}^{\text{ret}} & := & \{P\}\vdash\vec{c_b}; e? \\
 & & \mid & \forall(x:A).\ p_{\vdash}^{\text{return}} \\
\text{Simple full path}: & p_{s\vdash\dashv} & := & \{\}\vdash\vec{c_b}\dashv\{\} \\
\text{Complete full path}: & p_{\vdash\dashv} & := & \{P_1\}\vdash\vec{c_b}\dashv\{P_2\} \\
 & & \mid & \forall(x:A).\ p_{\vdash\dashv} \\
\text{Split result}: & \left\{ \begin{array}{l} \boldsymbol{p}_{\_}^{\text{nor}}, \boldsymbol{p}_{\_}^{\text{brk}}, \boldsymbol{p}_{\_}^{\text{con}}, \boldsymbol{p}_{\_}^{\text{ret}}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv} \end{array} \right\} & &
\end{array}
$$

Figure 3–2 syntax of split results

3.2 Split result interface

We define the syntax of the result of path splitting in Figure 3–2. The split result is a record type, consisting of "paths", "partial paths" and "atoms", which are essentially a list of basic program statements $\vec{c_b}$ annotated with two assertions, one single assertion, and no assertions, respectively. ① The basic statement can either be a primary Clight statement c_p or an expression e that introduces if conditions into the control flow.

Recall that in VST program logic, a statement has multiple post-conditions (exited normally, with break, continue, or return with a value). Correspondingly, our split result also makes distinctions between different exit status. The split result is defined as a record with 10 fields, constituted by one "full path", one "partial path" that only has post-condition, four "partial paths" that only have pre-condition and exit with four kinds of exit status, and four "atom paths" that exit with four kinds of exit statuses. ② Return atom paths and partial paths are augmented with an expression $?e$ that the program may return. With these fields, the split result record is sufficient to reveal all control flow information in a Clight-A program.

Similar to Clight-A syntax C_c, the split result is also parameterized by a simpler syntax in implementation. For simple results, the assertion is treated simply as a placeholder and there are no logical variables involved. For full results, each result type is a Coq Inductive type with an

① We use $\vec{c}$ to denote an ordered list of items of the syntax c. When the order is not important, as we shall see notations like $\boldsymbol{p}_{\dashv}$ in the split result record, we use the bold font to denote a set of items.

② For the sake of simplicity, we will refer to "partial paths" that only have pre-conditions as "post-partial paths", and "partial paths" that only have post-conditions as "pre-partial paths". For atom paths and post-partial paths, there are four kinds of exit statuses. We will refer to them by adding a prefix of the exit status name to distinguish them. For example, normal post-partial paths are partial-paths with only post-conditions and exit in normal status.

extra constructor to introduce logical variables into assertions. A complete result element p is in the dependepent type on a simple result p_s if p and p_s have the same list of basic statements.

Again, we will show how this dependepent relation is useful in the splitting function implementation in Coq in Section 3.4.

3.3 Interpreting split result

The split result we have defined above is a collection of basic program statement sequences, probably with assertions at the beginning or in the end. In practical settings, every program to verify will come with a pre-conditon and a post-condition specified by the user. By supplementing "partial paths"or "atoms"with the pre-/post-condition from the user's specification, we can interpret the split result into a collection of closed Hoare triples as verification goals.

To reuse the verification infrastructures, we choose to reinterpret the split result into the Hoare triples defined by VST. We implement the semantics interpretation procedure as Coq functions in Figure 3–3.

First, the to_Cstm function converts sequences of basic program statements into the abstract C language. Note that the basic statements of split results no longer contain control flow instructions such as break and continue, so for any to_Cstm($\vec{c_b}$), we only care about the case when it exits with normal status. We assign a True assertion $\top$ to post-condition fields other than normal. Besides, we make use of this observation to interpret the conditional expression e. We simply encode it into a simple if-branching structure that exit with break if e evaluates to false. This encoding is safe since the break post-condition is always true, and the path in question only cares about the case where e evaluates to true.

For interpreting pre-partial paths, semax_pre takes a user-specified pre-condition P, and interprets the result as a Hoare triple. This procedure is similar for semax_atom , the base case for semax_post, and the base case for semax_path. For logical variables syntactically binded to post-partial paths or full-paths, the interpretation function will convert them into a Coq-level universally quantification.

For return post-partial paths, since the post condition for return is parameterized by the return value. To match the VST program logic, we still make use of the return?e statement and the returning post condition in VST during interpretation. Now we only care about the control flow that exits with return status, so other post-conditions are set as $\top$.

Finally, the semax_split function takes a split result and interprets it into a conjunction of Hoare triples. All the interpretation functions are implemented in Coq as functions with Prop as the return type, which are left for users as the verification goals towards program correctness.

3.4 The split function

The core splitting function is a recursively defined Fixpoint function in Coq. It takes a Clight-A program as input and returns its split result in a record. Note that error may occur if there is a control flow in the loop that goes through no annotations, so in practice the function return type is the option type of the split result. We list the splitting function in Figure 3–4 and 3–5, and present a detailed explanation below.

$$
\begin{aligned}
&\mathsf{to\_Cstm}([]) &&= \mathsf{skip}\\
&\mathsf{to\_Cstm}(c_p :: \vec{c'_b}) &&= c_p; \mathsf{to\_Cstm}(\vec{c'_b})\\
&\mathsf{to\_Cstm}(e :: \vec{c'_b}) &&= \mathsf{if}\ (e)\ \mathsf{skip}\ \mathsf{else}\ \mathsf{break};\\
& && \quad \mathsf{to\_Cstm}(\vec{c'_b})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{semax\_pre}(P, \vec{c_b}\dashv\{Q\}) &= \{P\}\ \mathsf{to\_Cstm}(\vec{c_b})\ \{Q, [\vec{\top}]\}\\
\mathsf{semax\_atom}(P, Q, \vec{c_b}) &= \{P\}\ \mathsf{to\_Cstm}(\vec{c_b})\ \{Q, [\vec{\top}]\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{semax\_post}(Q, \{P\}\vdash\vec{c_b}) &&= \{P\}\ \mathsf{to\_Cstm}(\vec{c_b})\ \{Q, [\vec{\top}]\}\\
&\mathsf{semax\_post}(Q, \forall x.\ p_{\vdash}) &&= \forall x.\ \mathsf{semax\_post}(Q, p_{\vdash})\\
&\mathsf{semax\_path}(\{P\}\vdash\vec{c_b}\dashv\{Q\}) &&= \{P\}\ \mathsf{to\_Cstm}(\vec{c_b})\ \{Q, [\vec{\top}]\}\\
&\mathsf{semax\_path}(\forall x.\ p_{\mapsto}) &&= \forall x.\ \mathsf{semax\_path}(p_{\mapsto})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{semax\_atom\_ret}(P, Q, \vec{c_b}; ?e)\\
=\ &\{P\}\ \mathsf{to\_Cstm}(\vec{c_b}); \mathsf{return}\ e?\ \{\top, [\top, \top, Q]\}\\
&\mathsf{semax\_post\_ret}(Q, \{P\}\vdash\vec{c_b}; ?e)\\
=\ &\{P\}\ \mathsf{to\_Cstm}(\vec{c_b}); \mathsf{return}\ e?\ \{\top, [\top, \top, Q]\}\\
&\mathsf{semax\_post\_ret}(Q, \forall x.\ p^{\mathrm{return}}_{\mathrm{post}})\\
=\ &\forall x.\mathsf{semax\_post\_ret}(Q, p^{\mathrm{return}}_{\mathrm{post}})
\end{aligned}
$$

$$
\mathsf{semax\_split}\left(\begin{array}{l} P, Q, Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}},\\ \left\{\begin{array}{l} \boldsymbol{p}^{\mathrm{nor}}_{-}, \boldsymbol{p}^{\mathrm{brk}}_{-}, \boldsymbol{p}^{\mathrm{con}}_{-}, \boldsymbol{p}^{\mathrm{ret}}_{-},\\ \boldsymbol{p}^{\mathrm{nor}}_{\vdash}, \boldsymbol{p}^{\mathrm{brk}}_{\vdash}, \boldsymbol{p}^{\mathrm{con}}_{\vdash}, \boldsymbol{p}^{\mathrm{ret}}_{\vdash},\\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\mapsto} \end{array}\right\} \end{array}\right)
$$

$$
\begin{aligned}
=\ &\mathsf{Forall}\ (\mathsf{semax\_atom}\ P\ Q)\ \boldsymbol{p}^{\mathrm{nor}}_{-}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_atom}\ P\ Q_{\mathrm{brk}})\ \boldsymbol{p}^{\mathrm{brk}}_{-}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_atom}\ P\ Q_{\mathrm{con}})\ \boldsymbol{p}^{\mathrm{con}}_{-}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_atom\_ret}\ P\ Q_{\mathrm{ret}})\ \boldsymbol{p}^{\mathrm{ret}}_{-}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_post}\ Q)\ \boldsymbol{p}^{\mathrm{nor}}_{\vdash}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_post}\ Q_{\mathrm{brk}})\ \boldsymbol{p}^{\mathrm{brk}}_{\vdash}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_post}\ Q_{\mathrm{con}})\ \boldsymbol{p}^{\mathrm{con}}_{\vdash}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_post\_ret}\ Q_{\mathrm{ret}})\ \boldsymbol{p}^{\mathrm{ret}}_{\vdash}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_pre}\ P)\ \boldsymbol{p}_{\dashv}\\
\wedge\ &\mathsf{Forall}\ (\mathsf{semax\_path})\ \boldsymbol{p}_{\mapsto}
\end{aligned}
$$

Figure 3–3 semantics of split results

$$\text{split } c_p = \left\{ \begin{array}{l} \{[c_p]\}, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array} \right\} \quad \text{split break} = \left\{ \begin{array}{l} \emptyset, \{[]\}, \emptyset, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array} \right\}$$

$$\text{split continue} = \left\{ \begin{array}{l} \emptyset, \emptyset, \{[]\}, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array} \right\} \quad \text{split (return } e?) = \left\{ \begin{array}{l} \emptyset, \emptyset, \emptyset, \{([]; ?e)\} \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array} \right\}$$

$$\text{split (assert } P) = \left\{ \begin{array}{l} \emptyset, \emptyset, \emptyset, \emptyset \\ \{\{P\} \vdash []\}, \emptyset, \emptyset, \emptyset \\ \{[] \vdash \{P\}\}, \emptyset \end{array} \right\}$$

Figure 3–4 Split function for basic statements

Split result of primary statements For primary statements, only the normal atom path is a singleton of the statement itself. The other fields are left as empty set. For control flow related statements, their corresponding atom paths are a singleton of an empty list of basic statements, which will be interpreted as skip. Note the difference between empty set and singleton of an empty list. The empty set indicates that no possible executions will fall into the field's corresponding exit status, while the singleton of an empty list indicates that there exists exactly one possible control flow (which is simply skip and does nothing) in the split result. For the assertion constructor assert P, the split result will be a singleton of pre-partial path with P as its post-condition and a singleton of normal post-partial path with P as its pre-condition.

Combining split results The splitting of compound statements is the key of the splitting function. There is a "connect" operation $\cdot$ for sequencing two paths (probably with assertions in the head or tail). We abuse this notation to connect two set of paths, which works like multiplication, to pointwisely sequence each path in the two set, so that all possible control flow paths from two subprograms can be captured in the combined result. We use $+$ to union two sets of paths.

For splitting $c_1; c_2$, we first split c_1 and c_2 into paths $\boldsymbol{p}$'s and $\boldsymbol{q}$'s[①], and then construct the split result of $c_1; c_2$ by connecting corresponding control flow paths in $\boldsymbol{p}$'s and $\boldsymbol{q}$'s. For example, the break post-partial paths have three components, the ones from splitting c_1 ($\boldsymbol{p}^{\text{brk}}_{\vdash}$), the ones from splitting c_2 ($\boldsymbol{q}^{\text{brk}}_{\vdash}$), and the sequencing of the normal post-partial paths in c_1 and the break atom paths in c_2 ($\boldsymbol{p}^{\text{nor}}_{\vdash} \cdot \boldsymbol{q}^{\text{brk}}_{-}$).

The splitting of if-branching is simply adding the conditional expression to the head of all pre-partial paths and atom paths, and then union two groups of split results together. Since pre-partial paths and atom paths do not have assertions in the head, we can ensure that they capture all the possible execution paths where the conditional expression should be added.

The split result of loop (c_2) c_1 is more complicated. We need to restructure the control flow results related to break and continue in the loop body. First, since all break or continue related paths in the loop body will be digested by the loop structure, in the resulting split result, there are no (post-partial or atom) paths that exit with break or continue statuses. Next, the break post-partials

① In later sections, without special declaration, when two records of split results are involved, we will refer to contents of the first as $\boldsymbol{p}$ with subscripts, and the second as $\boldsymbol{q}$ with subscripts.

$$\text{let split } c_1 = \left\{ \begin{array}{l} \boldsymbol{p}_-^{\text{nor}}, \boldsymbol{p}_-^{\text{brk}}, \boldsymbol{p}_-^{\text{con}}, \boldsymbol{p}_-^{\text{ret}}, \\ \boldsymbol{p}_\vdash^{\text{nor}}, \boldsymbol{p}_\vdash^{\text{brk}}, \boldsymbol{p}_\vdash^{\text{con}}, \boldsymbol{p}_\vdash^{\text{ret}}, \\ \boldsymbol{p}_\dashv, \boldsymbol{p}_\mapsto \end{array} \right\} \text{ and split } c_2 = \left\{ \begin{array}{l} \boldsymbol{q}_-^{\text{nor}}, \boldsymbol{q}_-^{\text{brk}}, \boldsymbol{q}_-^{\text{con}}, \boldsymbol{q}_-^{\text{ret}}, \\ \boldsymbol{q}_\vdash^{\text{nor}}, \boldsymbol{q}_\vdash^{\text{brk}}, \boldsymbol{q}_\vdash^{\text{con}}, \boldsymbol{q}_\vdash^{\text{ret}}, \\ \boldsymbol{q}_\dashv, \boldsymbol{q}_\mapsto \end{array} \right\}$$

$$\text{split } (c_1; c_2) = \left\{ \begin{array}{l} \boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{nor}}, \\ \boldsymbol{p}_\vdash^{\text{brk}} +\!\!+ \boldsymbol{q}_\vdash^{\text{brk}} +\!\!+ \boldsymbol{p}_\vdash^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{brk}}, \\ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \boldsymbol{q}_\vdash^{\text{con}} +\!\!+ \boldsymbol{p}_\vdash^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{con}}, \\ \boldsymbol{p}_\vdash^{\text{ret}} +\!\!+ \boldsymbol{q}_\vdash^{\text{ret}} +\!\!+ \boldsymbol{p}_\vdash^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{ret}}, \\ \boldsymbol{p}_-^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{nor}}, \\ \boldsymbol{p}_-^{\text{brk}} +\!\!+ \boldsymbol{p}_-^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{brk}}, \\ \boldsymbol{p}_-^{\text{con}} +\!\!+ \boldsymbol{p}_-^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{con}}, \\ \boldsymbol{p}_-^{\text{ret}} +\!\!+ \boldsymbol{p}_-^{\text{nor}} \cdot \boldsymbol{q}_-^{\text{ret}}, \\ \boldsymbol{p}_\dashv +\!\!+ \boldsymbol{p}_-^{\text{nor}} \cdot \boldsymbol{q}_\dashv, \\ \boldsymbol{p}_\mapsto +\!\!+ \boldsymbol{q}_\mapsto \end{array} \right\}$$

$$\text{split } (\text{if } (e) \ c_1 \text{ else } c_2) = \left\{ \begin{array}{l} \boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{q}_\vdash^{\text{nor}} \\ \boldsymbol{p}_\vdash^{\text{brk}} +\!\!+ \boldsymbol{q}_\vdash^{\text{brk}} \\ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \boldsymbol{q}_\vdash^{\text{con}} \\ \boldsymbol{p}_\vdash^{\text{ret}} +\!\!+ \boldsymbol{q}_\vdash^{\text{ret}} \\ \{[e]\} \cdot \boldsymbol{p}_-^{\text{nor}} +\!\!+ \{[\neg e]\} \cdot \boldsymbol{q}_-^{\text{nor}} \\ \{[e]\} \cdot \boldsymbol{p}_-^{\text{brk}} +\!\!+ \{[\neg e]\} \cdot \boldsymbol{q}_-^{\text{brk}} \\ \{[e]\} \cdot \boldsymbol{p}_-^{\text{con}} +\!\!+ \{[\neg e]\} \cdot \boldsymbol{q}_-^{\text{con}} \\ \{[e]\} \cdot \boldsymbol{p}_-^{\text{ret}} +\!\!+ \{[\neg e]\} \cdot \boldsymbol{q}_-^{\text{ret}} \\ \{[e]\} \cdot \boldsymbol{p}_\dashv +\!\!+ \{[\neg e]\} \cdot \boldsymbol{q}_\dashv, \\ \boldsymbol{p}_\mapsto +\!\!+ \boldsymbol{q}_\mapsto \end{array} \right\}$$

$$\text{split } (\text{loop } (c_2) \ c_1) = \quad \text{if } (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{nor}} \neq \emptyset \text{ then Error}$$

$$\text{else} \left\{ \begin{array}{l} \boldsymbol{p}_\vdash^{\text{brk}} +\!\!+ \boldsymbol{q}_\vdash^{\text{brk}} +\!\!+ (\boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \boldsymbol{q}_\vdash^{\text{nor}} \cdot (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{brk}} \\ \qquad +\!\!+ \ (\boldsymbol{q}_\vdash^{\text{nor}} +\!\!+ (\boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{nor}}) \cdot \boldsymbol{p}_-^{\text{brk}}, \\ \emptyset, \\ \emptyset, \\ \boldsymbol{p}_\vdash^{\text{ret}} +\!\!+ \boldsymbol{q}_\vdash^{\text{ret}} +\!\!+ (\boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \boldsymbol{q}_\vdash^{\text{nor}} \cdot (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}})) \cdot \boldsymbol{q}_-^{\text{ret}} \\ \qquad +\!\!+ \ (\boldsymbol{q}_\vdash^{\text{nor}} +\!\!+ (\boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{nor}}) \cdot \boldsymbol{p}_-^{\text{ret}}, \\ \boldsymbol{p}_-^{\text{brk}} +\!\!+ (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{brk}}, \\ \emptyset, \\ \emptyset, \\ \boldsymbol{p}_-^{\text{ret}} +\!\!+ (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}}) \cdot \boldsymbol{q}_-^{\text{ret}}, \\ \boldsymbol{p}_\dashv +\!\!+ (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}}) \cdot (\boldsymbol{q}_\dashv +\!\!+ \boldsymbol{q}_-^{\text{con}} \cdot \{[]\vdash\{\bot\}\}), \\ \boldsymbol{p}_\mapsto +\!\!+ \boldsymbol{q}_\mapsto +\!\!+ \boldsymbol{q}_\vdash^{\text{nor}} \cdot \boldsymbol{p}_\dashv +\!\!+ \boldsymbol{q}_\vdash^{\text{con}} \cdot \{[]\vdash\{\bot\}\}) \\ \quad +\!\!+ \ (\boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \boldsymbol{q}_\vdash^{\text{nor}} \cdot (\boldsymbol{p}_-^{\text{nor}} +\!\!+ \boldsymbol{p}_-^{\text{con}})) \\ \qquad \cdot (\boldsymbol{q}_\dashv +\!\!+ \boldsymbol{q}_-^{\text{nor}} \cdot \boldsymbol{p}_\dashv +\!\!+ \boldsymbol{q}_-^{\text{con}} \cdot \{[]\vdash\{\bot\}\}) \end{array} \right\}$$

$$\text{split } (\text{ExGiven } x : A, \ \{P(x)\} \ c_1) = \left\{ \begin{array}{l} \emptyset, \emptyset, \emptyset, \emptyset \\ \boldsymbol{p}_\vdash^{\text{nor}} +\!\!+ \{\{P\}\vdash[]\} \cdot \boldsymbol{p}_-^{\text{nor}} \\ \boldsymbol{p}_\vdash^{\text{brk}} +\!\!+ \{\{P\}\vdash[]\} \cdot \boldsymbol{p}_-^{\text{brk}} \\ \boldsymbol{p}_\vdash^{\text{con}} +\!\!+ \{\{P\}\vdash[]\} \cdot \boldsymbol{p}_-^{\text{con}} \\ \boldsymbol{p}_\vdash^{\text{ret}} +\!\!+ \{\{P\}\vdash[]\} \cdot \boldsymbol{p}_-^{\text{ret}} \\ \{[]\vdash\{\exists x : A. \ P(x)\}\}, \\ \boldsymbol{p}_\mapsto +\!\!+ \{\{P\}\vdash[]\} \cdot \boldsymbol{p}_\dashv \end{array} \right\}$$

Figure 3–5 Split function for compound statements

($\boldsymbol{p}_{\vdash}^{\text{brk}}$, $\boldsymbol{q}_{\vdash}^{\text{brk}}$) in the two loop bodies become normal post-partials when viewed outside the loop. The break atom paths ($\boldsymbol{p}_{\_}^{\text{brk}}$, $\boldsymbol{q}_{\_}^{\text{brk}}$) in the two loop bodies will be combined with all possible pre-conditions in the loop and also become part of normal post-partial paths and normal atom paths. The case for return related post-partial/atom paths works in a similar way as the case for break. For continue paths, those in the main loop body c_1 are processed similarly as normal paths - they will both execute the incremental loop body c_2 next, while for continue paths in the incremental loop body c_2, they are disallowed to be executed, so a false post-condition $\bot$ is added to continue post-partial/atom paths.

Dealing with logical variables To split ExGiven $x : A$, $\{P(x)\}\ c'$, the new variable x will be treated differently in pre-partial paths and post-partial paths. For pre-partial paths, since program assertions that appear before the ex-given structure will not mention the new variable x, it is safe to move x into the assertion P as an existential quantifier, leaving the pre-partial paths to be a singleton of $[]\dashv\{\exists x : A.\ P(x)\}$. There are no atom paths in the split result of ex-given structures, since the strucutre itself ensure that a pre-condition $P(x)$ appear in the head of the program. Post-partial paths for each exit status have two components, the post-partial paths of the inner statement c' (where x in c will be quantified universally), and the pre-condition $P(x)$ sequenced with atom-paths that exit with the corresponding status. For full paths, in addition to the closed full paths that are already split and collected in c', we also need to sequence the pre-condition $P(x)$ to the pre-partial paths in c'. Note that this connecting operation requires unifying the existential variable x with the one in pre-partial paths split from c, so that the logical variable x can be shared among the pre-/post-conditions of each combined full path.

A technical issue arises in implementing the split function for ex-given structure. We encode the quantification of logical variables as function types in Coq as follows

```
Inductive C_statement' : Type :=
| Cexgiven' (A:Type)
    (ass: A -> assert) (stm': A -> C_statement')
| ...
.
```

If we were to implement a recursive splitting function on C_statement', we can at most get the split result of the inner statement stm' in the form of

```
(fun a => split (stm' a)) : A -> split_result.
```

However, to construct the split result of ex-given structure, we need to extract each individual split result that are of the type A -> some_path from the the abstracted inner split result that is of the type A -> split_result.

To address this issue, we design the Clight-A syntax and the split result interface to be of Coq dependepent types on simpler syntaxes. The simpler syntaxes erase the logical variables and preserve only the shape of the Clight-A program or the split result. Now the Clight-A syntax and split function signature in Coq are as follows.

```
Inductive S_statement : Type :=
| Sassert
```

```
| Ssequence (c1 c2 : S_statement)
| ...
.

Inductive C_statement : S_statement -> Type :=
| Cexgiven: forall (A:Type)
    (ass: A -> assert)
    (c: S_statement)
    (stm': A -> C_statement c),
    C_statement ((Ssequence Sassert c ))
| Cassert : assert -> C_statement Sassert
| ...
.

Inductive S_result : Type := ... .

Inductive C_result : S_result -> Type := ... .

Fixpoint S_split (s: S_statement) : S_result.

Fixpoint C_split (s: S_statement)
    (c: C_statement s) : C_result (S_split s).
```

Now, the split result of inner statement c' is dependepent on a simpler result that is not abstracted by the logical variable's type A, we are able to perform pattern-matching on the simpler result, and extract individual paths out from the packed split result abstracted by A.

3.5 Soundness

To ensure end-to-end corretness of our framework, we formalized the soundness theorem of the split function in Coq based on VST. The soundness theorem states that, given any program with its pre-/post-condition, if all Hoare triples in its split result can be verified, then the original statement should be provable.

Theorem 3.1 (Soundness) For any Clight-A program p, pre-condition P and post-conditions $\vec{Q}$, if split $p \neq$ Error and semax_split$(P, \vec{Q}, \text{split } p)$ holds, then Hoare triple $\{P\}\ p\ \left\{\vec{Q}\right\}$ holds. ①.

The soundness theorem is proved by induction on the statement to be split. Almost all of the soundness proof can be done on a logical level except one lemma (the conjunction rule, which will be proved in Section 4). The Hoare logic rules we uses are presented in Figure 2–2 and 2–3. We use $\Sigma; \Gamma$ to represent the proof context of logical variables and pure propositions that can be used to

① In the Hoare triple, all assertions in p are removed to match the Clight syntax. The translation is straightforward, simply by replacing assertions with a skip command in the Clight language.

derive Hoare triples in the logical rules, but we will omit them in following text since they stays the same or the change of proof context is easy to be recognized.

The rules are formalized in a deeply embedded fashion, i.e. as an Inductive relation in Coq. The benefit of deeply embedded Hoare logic formalization is that it makes a rich bundle of derived rules readily available, a few of which are shown in Figure 2–4. We also make use of the following inversion lemmas when proving the soundness of our splitting function.

Lemma 3.1 (Inversion on sequencing) If $\{P\}\, c_1; c_2\, \left\{Q, [\vec{Q'}]\right\}$, then

$$\{P'\}\, c_1\, \left\{\exists R : \mathsf{assert}.\ R \wedge \{R\}\, c_2\, \left\{Q, [\vec{Q'}]\right\}, [\vec{Q'}]\right\}$$

Lemma 3.2 (Inversion on if-branching) If $\{P\}\, \mathsf{if}\ (b)\ c_1\ \mathsf{else}\ c_2\, \left\{Q, [\vec{Q'}]\right\}$, then

$$\begin{aligned} P \vDash \exists P' : \mathsf{assert},\ & P' \\ & \wedge \{P' \wedge [\![b]\!] = \mathsf{true}\}\, c_1\, \left\{Q, [\vec{Q'}]\right\} \\ & \wedge \{P' \wedge [\![b]\!] = \mathsf{false}\}\, c_2\, \left\{Q, [\vec{Q'}]\right\} \end{aligned}$$

Lemma 3.3 (Inversion on skip) If $\{P\}\, \mathsf{skip}\, \left\{Q, [\vec{Q'}]\right\}$, then $P \vDash Q$

Lemma 3.4 (Inversion on break) If $\{P\}\, \mathsf{break}\, \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}$, then $P \vDash Q_{\mathrm{brk}}$

The split results of basic operations are sound by definition. In fact, during the proof, we do not even need to look at the detailed rules for primary statements, since we choose to interpret the split result in the same way as the original statement. For control flow statements such as break, applying the inversion Lemma 3.3 to the corresponding control flow path can complete the proof. The rest of this section will focus on proving the soundness for compound statements.

Weakest pre-condition from inversion lemmas Consider the soundness of splitting sequential composition $c_1; c_2$. By the rule SEMAX-SEQ, we need to find an intermediate assertion R to derive $\{P\}\, c_1; c_2\, \left\{\vec{Q}\right\}$. Intuitively, the intermediate assertion R should be the conjunction of the strongest weakest pre-conditions of each control flow in the second statement.

The VST higher order assertion allows us to write an explicit representation of weakest pre-condition in the assertion language, justified by the following lemma.

Lemma 3.5 For any program c and post-condition $\vec{Q}$, it holds that $\{\exists P : \mathsf{assert}.\ P \wedge \{P\}\, c\, \left\{\vec{Q}\right\}\}\, c\, \left\{\vec{Q}\right\}$.

If we take a closer look at Lemma 3.1, it is essentially stating that the weakest pre-condition of the second statement can be used to serve as the intermediate assertion for sequential composition. Based on Lemma 3.1, we can prove for each type of paths in the split results a corresponding inversion lemma on the $\cdot$ operator.

Proposition 3.1 (Inversion lemmas for split results)

1. If $\mathsf{semax\_atom}(P, Q, p_- \cdot q_-)$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_atom}(R, Q, q_-), p_-)$

2. If $\mathsf{semax\_atom\_ret}(P, Q, p_- \cdot q_-^{\mathrm{ret}})$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_atom\_ret}(R, Q, q_-^{\mathrm{ret}}), p_-)$
3. If $\mathsf{semax\_pre}(P, p_- \cdot q_\dashv)$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_pre}(R, q_\dashv), p_-)$
4. If $\mathsf{semax\_post}(Q, p_\vdash \cdot q_-)$, then
   $\mathsf{semax\_post}(Q, \exists R.\ R \wedge \mathsf{semax\_atom}(R, Q, q_-), p_\vdash)$
5. If $\mathsf{semax\_post\_ret}(Q, p_\vdash \cdot q_-^{\mathrm{ret}})$, then
   $\mathsf{semax\_post}(Q, \exists R.\ R \wedge \mathsf{semax\_atom\_ret}(R, Q, q_-^{\mathrm{ret}}), p_\vdash)$
6. If $\mathsf{semax\_path}(p_\vdash \cdot q_\dashv)$, then
   $\mathsf{semax\_post}(\exists R.\ R \wedge \mathsf{semax\_pre}(R, q_\dashv), p_\vdash)$

The inversion lemma also holds for the case where the first argument of $\cdot$ is a post-partial path with quantified structures. Consider item 4 in Proposition 3.1, for example, the proof can be done by induction on $p_\vdash$. When $p_\vdash = \forall x.\ p'_\vdash$, we know that $\forall x.\mathsf{semax\_post}(Q, p'_\vdash \cdot q_-)$. The goal can be simplified as $\forall x.\mathsf{semax\_post}(Q, \exists R.\ R \wedge \mathsf{semax\_atom}(R, Q, q_-), p'_\vdash)$, which can be proved directly from the induction hypothesis.

Combining weakest pre-conditions With Proposition 3.1, we can collect the intermediate assertions for each control flow that crosses the two sub-statements, but to apply the logical rules for the original Clight program, we need to combine all the intermediate assertions into one.

We start with combining intermediate assertions for the same type of paths in the split result. Note that the $\cdot$ operator works on a set of paths instead of individual path in the split function, so Proposition 3.1 should be extended to the following form ①

Proposition 3.2 (Grouped Inversion lemmas)

1. If $\boldsymbol{q}_- \neq \emptyset$ and $\mathsf{semax\_atom}(P, Q, \boldsymbol{p}_- \cdot \boldsymbol{q}_-)$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{q}_-), \boldsymbol{p}_-)$
2. If $\boldsymbol{q}_-^{\mathrm{ret}} \neq \emptyset$ and $\mathsf{semax\_atom\_ret}(P, Q, \boldsymbol{p}_- \cdot \boldsymbol{q}_-^{\mathrm{ret}})$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_atom\_ret}(R, Q, \boldsymbol{q}_-^{\mathrm{ret}}), \boldsymbol{p}_-)$
3. If $\boldsymbol{q}_\dashv \neq \emptyset$ and $\mathsf{semax\_pre}(P, \boldsymbol{p}_- \cdot \boldsymbol{q}_\dashv)$, then
   $\mathsf{semax\_atom}(P, \exists R.\ R \wedge \mathsf{semax\_pre}(R, \boldsymbol{q}_\dashv), \boldsymbol{p}_-)$
4. If $\boldsymbol{q}_- \neq \emptyset$ and $\mathsf{semax\_post}(Q, \boldsymbol{p}_\vdash \cdot \boldsymbol{q}_-)$, then
   $\mathsf{semax\_post}(Q, \exists R.\ R \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{q}_-), \boldsymbol{p}_\vdash)$
5. If $\boldsymbol{q}_-^{\mathrm{ret}} \neq \emptyset$ and $\mathsf{semax\_post\_ret}(Q, \boldsymbol{p}_\vdash \cdot \boldsymbol{q}_-^{\mathrm{ret}})$, then
   $\mathsf{semax\_post}(Q, \exists R.\ R \wedge \mathsf{semax\_atom\_ret}(R, Q, \boldsymbol{q}_-^{\mathrm{ret}}), \boldsymbol{p}_\vdash)$
6. If $\boldsymbol{q}_\dashv \neq \emptyset$ and $\mathsf{semax\_path}(\boldsymbol{p}_\vdash \cdot \boldsymbol{q}_\dashv)$, then
   $\mathsf{semax\_post}(\exists R.\ R \wedge \mathsf{semax\_pre}(R, \boldsymbol{q}_\dashv), \boldsymbol{p}_\vdash)$

Proposition 3.2 can be proved by induction fisrt on the size of set on the LHS of the $\cdot$ operator then on the size of set on the RHS. The induction on the LHS argument is straightforward, since the intermediate assertions we obtain from the inversion lemmas are the same if we fix the RHS

① We will abuse notations like $\mathsf{semax\_path}(\boldsymbol{p}_\mapsto)$ where $\boldsymbol{p}_\mapsto$ is a set of paths to represent (Forall semax_path $\boldsymbol{p}_\mapsto$) in the following thesis.

argument. Consider the case for the second induction when proving Proposition 3.2 (4). Assume $\boldsymbol{q}_- = q_- :: \boldsymbol{q}'_-$. Applying Proposition 3.1 (4) to q_- we get

$$\mathsf{semax\_post}(\exists R_1.\ \mathsf{semax\_pre}(R_1, q_-), \boldsymbol{p}_\vdash)$$

. The induction hypothesis gives

$$\mathsf{semax\_post}(\exists R_2.\ \mathsf{semax\_pre}(R_2, \boldsymbol{q}'_-), \boldsymbol{p}_\vdash)$$

. Let R be $R_1 \wedge R_2$, we can derive that

$$\begin{array}{rl} & \exists R_1.\ \mathsf{semax\_pre}(R_1, q_-) \wedge \exists R_2.\ \mathsf{semax\_pre}(R_2, \boldsymbol{q}'_-) \\ \vDash & \exists R.\ \mathsf{semax\_pre}(R, \boldsymbol{q}_-) \end{array}$$

To finish the proof, we need to have the following lemmas to enable conjunction on post-condition:

Proposition 3.3 (Conjunction rule on paths)

1. If $\mathsf{semax\_post}(Q_1, q_\vdash)$ and $\mathsf{semax\_post}(Q_2, q_\vdash)$, then $\mathsf{semax\_post}(Q_1 \wedge Q_2, q_\vdash)$
2. If $\mathsf{semax\_atom}(P, Q_1, q_-)$ and $\mathsf{semax\_atom}(P, Q_2, q_-)$, then $\mathsf{semax\_atom}(P, Q_1 \wedge Q_2, q_-)$
3. If $\mathsf{semax\_atom\_ret}(P, Q_1, q_-^{\mathrm{ret}})$ and $\mathsf{semax\_atom\_ret}(P, Q_2, q_-^{\mathrm{ret}})$, then $\mathsf{semax\_atom\_ret}(P, Q_1 \wedge Q_2, q_-^{\mathrm{ret}})$
4. If $\mathsf{semax\_post}(Q_1, \boldsymbol{q}_\vdash)$ and $\mathsf{semax\_post}(Q_2, \boldsymbol{q}_\vdash)$, then $\mathsf{semax\_post}(Q_1 \wedge Q_2, \boldsymbol{q}_\vdash)$
5. If $\mathsf{semax\_atom}(P, Q_1, \boldsymbol{p}_-)$ and $\mathsf{semax\_atom}(P, Q_2, \boldsymbol{p}_-)$, then $\mathsf{semax\_atom}(P, Q_1 \wedge Q_2, \boldsymbol{p}_-)$
6. If $\mathsf{semax\_atom\_ret}(P, Q_1, \boldsymbol{p}_-^{\mathrm{ret}})$ and $\mathsf{semax\_atom\_ret}(P, Q_2, \boldsymbol{p}_-^{\mathrm{ret}})$, then $\mathsf{semax\_atom\_ret}(P, Q_1 \wedge Q_2, \boldsymbol{p}_-^{\mathrm{ret}})$

The above propositions require the underlying program logic to be able to derive the *Conjunction Rule*, stated as follows.

Theorem 3.2 (Conjunction Rule) if Hoare triples $\{P\}\ c\ \left\{Q_1, [\vec{Q}'_1]\right\}$ and $\{P\}\ c\ \left\{Q_1, [\vec{Q}'_2]\right\}$ are derivable, then $\{P\}\ c\ \left\{Q_1 \wedge Q_2, [\vec{Q}'_1 \wedge \vec{Q}'_2]\right\}$ is derivable holds.

However, the conjunction rule is not ubiquitous among Hoare logic variants proposed in literatures. For example, the current VST program logic cannot derive the conjunction rule. We will leave the discussion of the conjunction rule to Section 4. For now, we assume that the conjunction rule is available, so that we construct the intermediate assertion for SEMAX-SEQ as follows:

$$R = \left\{ \begin{array}{l} \exists R.\ R \wedge \mathsf{semax\_pre}(R, \boldsymbol{q}_\dashv) \\ \quad \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{q}_-^{\mathrm{nor}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q_{\mathrm{brk}}, \boldsymbol{q}_-^{\mathrm{brk}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q_{\mathrm{con}}, \boldsymbol{q}_-^{\mathrm{con}}) \\ \quad \wedge \mathsf{semax\_atom\_ret}(R, Q_{\mathrm{ret}}, \boldsymbol{q}_-^{\mathrm{ret}}) \end{array} \right\}$$

We can next prove that

$$\mathsf{semax\_split}(P, R, Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}, \mathsf{split}\ c_1) \wedge \mathsf{semax\_split}(R, Q, Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}, \mathsf{split}\ c_2)$$

holds from the premise that $\mathsf{semax\_split}(P, Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}, \mathsf{split}\ c_1; c_2)$ using the inversion lemmas and the conjunction rule. The fact that our split function collects all the control flow paths separated by user-provided assertions guarantees that this operation is feasible. Next, by applying the induction hypothesis, we obtain that $\{P\}\ c_1\ \{R, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}$ and $\{R\}\ c_2\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}$ are derivable, which leads to the soundness for $\mathsf{split}\ c_1; c_2$

Soundness of splitting loop Similarly, for soundness of $\mathsf{split\ loop}\ (c_2)\ c_1$, we construct the loop invariant I and the invariant for the incremental step I_{con} required by Semax-Loop as follows:

$$I = \left\{ \begin{array}{l} \exists R.\ R \wedge \mathsf{semax\_pre}(R, \boldsymbol{p}_{\dashv}) \\ \quad \wedge \mathsf{semax\_pre}(R, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{\dashv}) \\ \quad \wedge \mathsf{semax\_atom}(R, \bot, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{con}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{p}_{-}^{\text{brk}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{brk}}) \\ \quad \wedge \mathsf{semax\_atom\_ret}(R, Q_{\text{ret}}, \boldsymbol{p}_{-}^{\text{ret}}) \\ \quad \wedge \mathsf{semax\_atom\_ret}(R, Q_{\text{ret}}, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{ret}}) \end{array} \right\}$$

$$I_{\text{con}} = \left\{ \begin{array}{l} \exists R.\ R \wedge \mathsf{semax\_pre}(R, \boldsymbol{q}_{\dashv}) \\ \quad \wedge \mathsf{semax\_pre}(R, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{\dashv}) \\ \quad \wedge \mathsf{semax\_atom}(R, \bot, \boldsymbol{q}_{-}^{\text{con}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{q}_{-}^{\text{brk}}) \\ \quad \wedge \mathsf{semax\_atom}(R, Q, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{-}^{\text{brk}}) \\ \quad \wedge \mathsf{semax\_atom\_ret}(R, Q_{\text{ret}}, \boldsymbol{q}_{-}^{\text{ret}}) \\ \quad \wedge \mathsf{semax\_atom\_ret}(R, Q_{\text{ret}}, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{-}^{\text{ret}}) \end{array} \right\}$$

By inversion from the premise that

$$\mathsf{semax\_split}(P, Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}, \mathsf{split}\ (\mathsf{loop}\ (c_2)\ c_1))$$

, we can also derive the two Hoare triples required by Semax-Loop and prove $\mathsf{split}\ (\mathsf{loop}\ (c_2)\ c_1)$ sound.

Soundness of if-branching The key for this case is to show that it is safe to translate the conditional expression statement into the pre-condition.

Recall the interpretation function we have defined for conditional expression:

$$\mathsf{to\_Cstm}([e]) = \mathsf{if}\ (e)\ \mathsf{skip}\ \mathsf{else}\ \mathsf{break}$$

. To translate from them, we make use of the inversion lemmas that are available in deep-embedded VST program logic. The following propositions can be derived from the Lemma 3.2, 3.3 and 3.4.

Proposition 3.4 (Conditional Expression lemmas)

1. If $\mathsf{semax\_pre}(P, \{[e]\} \cdot \boldsymbol{p}_{\dashv})$, then $\mathsf{semax\_pre}(P \wedge [\![e]\!] = \mathsf{true}, \boldsymbol{p}_{\dashv})$
2. If $\mathsf{semax\_atom}(P, Q, \{[e]\} \cdot \boldsymbol{p}_{-})$, then $\mathsf{semax\_atom}(P \wedge [\![e]\!] = \mathsf{true}, Q, \boldsymbol{p}_{-})$
3. If $\mathsf{semax\_atom\_ret}(P, Q, \{[e]\} \cdot \boldsymbol{p}_{-}^{\text{ret}})$, then $\mathsf{semax\_atom\_ret}(P \wedge [\![e]\!] = \mathsf{true}, Q, \boldsymbol{p}_{-}^{\text{ret}})$

We show the proof detail for item 1.

Proof We consider any pre-partial path $p_{\dashv}$ in $\boldsymbol{p}_{\dashv}$. From Proposition 3.1 (3), we have that

$$\{P\}\,\mathsf{to\_Cstm}([e])\ \{\exists R.\ R \wedge \mathsf{semax\_pre}(R, p_{\dashv}), [\vec{\perp}]\}$$

Applying lemmas 3.2, 3.3, and 3.4, we obtain

$$\begin{aligned} P \vDash \exists P' : \mathsf{assert},\ & P' \\ & \wedge (P' \wedge [\![e]\!] = \mathsf{true} \vDash \exists R.\ R \wedge \mathsf{semax\_pre}(R, p_{\dashv})) \\ & \wedge (P' \wedge [\![e]\!] = \mathsf{false} \vDash \perp) \end{aligned}$$

By Semax-Conseq, to prove $\mathsf{semax\_pre}(P \wedge [\![e]\!] = \mathsf{true}, p_{\dashv})$, we only need to show

$$\begin{aligned} &\text{for any } P', \text{ if} \\ &\quad P' \wedge [\![e]\!] = \mathsf{true} \vDash \exists R.\ R \wedge \mathsf{semax\_pre}(R, p_{\dashv}) \\ &\quad P' \wedge [\![e]\!] = \mathsf{false} \vDash \perp \\ &\text{then } \mathsf{semax\_pre}(P' \wedge [\![e]\!] = \mathsf{true}, p_{\dashv}) \end{aligned}$$

which follows directly from the fact that

$$\mathsf{semax\_pre}(\exists R.\ \mathsf{semax\_pre}(R, p_{\dashv})), p_{\dashv})$$

After transformation using Proposition 3.4 in the premise of the soundness theorem, the induction hypothesis can be directly applied to complete the proof. We omit the dual case where the $\neg e$ branch is taken.

Soundness of ex-given structure The premise that

$$\mathsf{semax\_split}(P_0, \vec{Q}, \mathsf{split}\ (\text{ExGiven}\ x : A,\ \{P(x)\}\ c'))$$

leads to the following two propositions.

1. $P_0 \vDash \exists x : A.\ P(x)$
2. $\forall x : A.\ \mathsf{semax\_split}(P(x), \vec{Q}, \mathsf{split}\ (c'))$

The first proposition can be obtained from the pre-partial path $\{[]\dashv\{\exists x : A.\ P(x)\}\}$. The second proposition can be obtained from the rest of the fields in the split result. The soundness proof is done by first by applying Semax-Conseq to transform the pre-condition of the triple to be $\exists x : A.\ P(x)$. Next, the Extract-Exists rule is used to introduce the existentially quantified x into the proof context. Then the induction hypothesis and the second proposition above are applied to complete the proof.

3.6 Summary

In this chapter, we present Clight-A, a formal language for annotated C program, and also the syntax of split results respectively in §3.1 and §3.2. We also introduce how split results are interpreted as verification goals in §3.3. The core of the VST-A framework is a split function that takes an Clight-A program and returns a set of split results. We define the split function and provide some explanations in §3.4. We also show that the split function is sound in §3.5. During the proof, we need to use the conjunction rule, which will be discussed in the next chapter.

Chapter 4 Conjunction rule

The key missing ingredient of the soundness proof in the previous section is the conjunction rule (Theorem 3.2).

In this section we will study more closely into why the conjunction rule is required by the soundness of our splitting algorithm. To make conjunction rule available in VST-A, we identify a new notion of preciseness to restrict the function specifications being called during verification. We will also discuss the trade-offs of using conjunction rules and precise function specifications, and suggest some future directions to improve on them.

4.1 Why conjunction rule

The soundness proof of our splitting algorithm is performed on the logic level, but requires the conjunction rule to be available. The need for conjunction rule is not determined by the way we prove soundness, but by the framework we propose to first split the program into separate paths and then verify them separately.

Hoare logic with ghost variables and ghost updates serves as an illustration of why we need conjunction rule for the splitting algorithm's soundness. Ghost state is "logical state" that can be considered as essential to maintain the invariant of the program but not part of the program's state. Ghost states are particularily useful for verifying concurrent programs. Ghost states can be manipulated by ghost actions that are inserted into the program and make updates to ghost variables. Consider the splitting example in Section 1 for

```
/* P */ c1; if (b) { c2 /* Q */ c3 } else { c4 /* Q' */ c5 } /* R */
```

again. Note that c_1 may contain ghost updates, and to verify the two if-branches, different ghost actions can be inserted. In a full power interactive verification tool, such proof is feasible by doing meta-logic reasoning. For example, users may perform a Coq-level case analysis in VST to provide two distinct assertions T and T' after c_1 is executed from a state satisfying P. say

```
/* P */ c1 /* T */,   /* P */ c1 /* T' */
```

. The two triples can be proved by inserting c_1 with two different ghost actions. Clearly, the conjunction on post conditions for these two triples are not derivable, but if users can prove that T and T' ensures only one branch is taken, then they can just choose to use one of them when they prove the two if-branches. As a result, they are still able to prove the the program without the conjunction rule.

Such intricate proof strategy is not feasible in VST-A. It requires more manual efforts and is beyond the expressive power of writing assertion annotations in the program. The example also shows that if we want to derive the whole program's correctness from path verification results, we must disallow a path to have distinct ghost actions when combined with other different paths. Such restriction can be achieved by requiring the logic to derive the conjunction rule.

Despite the need for conjunction rule, we show that our framework is still powerful in verifying useful programs. To be specific, we base our VST-A framework on a a stronger (but still sound)

variant of the VST program logic in Figure 2–2 and Figure 2–3, where logical operators related to concurrency are removed for sake of deriving the conjunction rule, but users are still allowed to write unrestricted higher order predicates. We prove the conjunction rule in this stronger logic.

Theorem 4.1 (Conjunction Rule) If Hoare triples $\{P\}\, c\, \left\{Q_1, [\vec{Q}_1']\right\}$ and $\{P\}\, c\, \left\{Q_1, [\vec{Q}_2']\right\}$ are derivable, then $\{P\}\, c\, \left\{Q_1 \wedge Q_2, [\vec{Q}_1' \wedge \vec{Q}_2']\right\}$ is derivable.

The proof can be done by induction on the abstract syntax of statement c. For constructs that use rules in Figure 2–2, the proof is straightforward and can be done on the logic level.

4.2 Proving the conjunction rule in VST

The current VST program logic, as an impredicative higher order concurrent separation logic, does not imply the conjunction rule. To make the conjunction rule available in VST-A, the logical rules we have presented in Figure 2–2 and 2–3 is actually a stronger (but still sound) variant of the current VST program logic. To be specific, we remove logical operators related to concurrency, and add extra constraints to Semax-Call.

We present a proof draft for the conjunction rule of our stronger logical system below. We will leave some of the goals that require model-level insights of VST underlying semantics to the proceeding sections.

Theorem 4.2 (Conjunction Rule) if Hoare triples $\{P\}\, c\, \left\{Q_1, [\vec{Q}_1']\right\}$ and $\{P\}\, c\, \left\{Q_1, [\vec{Q}_2']\right\}$ are derivable, then $\{P\}\, c\, \left\{Q_1 \wedge Q_2, [\vec{Q}_1' \wedge \vec{Q}_2']\right\}$ is derivable.

Proof We prove by induction on the statement c.

- $c = \mathsf{skip}$. Applying Lemma 3.3 on premises we get $P \vDash Q_1$ and $P \vDash Q_2$. Thus, $P \vDash Q_1 \wedge Q_2$ is derivable, and the result follows from Semax-Skip. The proof is simiar when c is break, continue, and return?e.
- $c = c_1; c_2$. Applying Lemma 3.1 on premises we get

$$\begin{array}{l}\{P\}\, c_1\, \left\{\exists R_1.\ R_1 \wedge \{R_1\}\, c_2\, \left\{Q_1, [\vec{Q}_1']\right\}, [\vec{Q}_1']\right\} \\ \{P\}\, c_1\, \left\{\exists R_2.\ R_2 \wedge \{R_2\}\, c_2\, \left\{Q_2, [\vec{Q}_2']\right\}, [\vec{Q}_2']\right\}\end{array}$$

  We can apply the induction hypothesis of c_1 and make use of Semax-Conseq to obtain

$$\{P\} c_1 \left\{ \begin{array}{l} \exists R.\ R \\ \wedge \{R\}\, c_2\, \left\{Q_1, [\vec{Q}_1']\right\} \\ \wedge \{R\}\, c_2\, \left\{Q_2, [\vec{Q}_2']\right\} \end{array}, \left[\vec{Q}_1' \wedge \vec{Q}_2'\right] \right\}$$

  where R can be taken as $R_1 \wedge R_2$. According to Semax-Seq, we are left to prove

$$\left\{ \begin{array}{l} \exists R.\ R \\ \wedge \{R\}\, c_2\, \left\{Q_1, [\vec{Q}_1']\right\} \\ \wedge \{R\}\, c_2\, \left\{Q_2, [\vec{Q}_2']\right\} \end{array} \right\} c_2 \left\{ \begin{array}{l} Q_1 \wedge Q_2, \\ \left[\vec{Q}_1' \wedge \vec{Q}_2'\right] \end{array} \right\}$$

  Using Extract-Exists and Extract-Prop we can extract R and the two Hoare triples into the proof context. By applying the induction hypothesis of c_2 on the two Hoare triples, we can obtain the result.

- $c = \mathsf{if}\ (e)\ c_1\ \mathsf{else}\ c_2$. By Lemma 3.2, we can obtain R_1 and R_2 and the following triples in the proof context:

$$\{R_1 \wedge [\![e]\!] = \mathsf{true}\}\ c_1\ \left\{Q_1, [\vec{Q}'_1]\right\}$$
$$\{R_1 \wedge [\![e]\!] = \mathsf{false}\}\ c_2\ \left\{Q_1, [\vec{Q}'_1]\right\}$$
$$\{R_2 \wedge [\![e]\!] = \mathsf{true}\}\ c_1\ \left\{Q_2, [\vec{Q}'_2]\right\}$$
$$\{R_2 \wedge [\![e]\!] = \mathsf{false}\}\ c_2\ \left\{Q_2, [\vec{Q}'_2]\right\}$$

  Both R_1 and R_2 are derivable from P, and the result follows by applying the induction hypothesis on respective triples.

- $c = \mathsf{loop}\ (c_2)\ c_1$. The inversion lemma for loop structures will provide two pair of loop invariants I_1, I_1^{incr} and I_2, I_2^{incr} for the two premises respectively, where I_1 and I_2 are derivable from P, together with the following triples available:

$$\{I_1\}\ c_1\ \left\{I_1^{incr}, [Q_1, I_1^{incr}, Q'_{ret1}]\right\}$$
$$\{I_1^{incr}\}\ c_2\ \left\{I_1, [Q_1, \bot, Q'_{ret1}]\right\}$$
$$\{I_2\}\ c_1\ \left\{I_2^{incr}, [Q_2, I_2^{incr}, Q'_{ret2}]\right\}$$
$$\{I_2^{incr}\}\ c_2\ \left\{I_2, [Q_2, \bot, Q'_{ret2}]\right\}$$

  Similarly, we can strengthen the pre-condition to be $I_1 \wedge I_2$ and $I_1^{incr} \wedge I_2^{incr}$ respectively, so that we can use the induction hypothesis to combine the two triples for c_1 and c_2 into one, which completes the proof. □

Above, we have proved most of the inductive cases for the conjunction rule, except the cases where the statement c belongs to primary statements, namely memory loading, memory writing and function calls. Proving the conjunction rule for these base cases is non-trivial, since VST is a rich logical system with support for fractional permissions[25, 26] and function calls with subsumptions[27]. Attempts to prove these cases on logical level have failed. We need model-level insights into VST underlying semantics. The rest of this section will demonstrate that the conjunction rule still holds under the settings of VST, so that assertions in VST-A can enjoy the rich features provided by VST. During the proof, we will also identify a key property, which is referred to as "preciseness", to be necessary for primary operations to derive the conjunction rule.

4.3 Conjunction rule for memory load

For primary statements like assignment and function calls, in previous parts of this thesis, we do not care about their semantics. Since our framework interpret them in the same way as the original program, the concrete semantics do not affect the soundness of the split function, as long as the program logic provides the conjunction rule. However, to prove the conjunction rule, we need to take a closer look.

The logical rules for primary statements have been presented in Figure 2–3. We begin with rules related to assignment statements $e_1 := e_2$.

We use $p \mapsto_\pi v$ to describe a singleton heap containing value v at address p, with permission-share π. The predicate can correspond to either the "address_mapsto" or the "mapsto" predicate (the derived form of the former) in VST, depending on the reasoning level. In VST, the $\mapsto$ predicate

is also parameterized with the C-type of v to enable C-type checking verification. We will abuse the notation of mapsto($\mapsto$), and omit the type-checking parameter in this thesis.

The $\triangleright P$ predicate is a higher-order separation logic operator saying that instead of P we have a slightly weaker approximation to it. The theory of higher-order separation logic has been well studied, and the addition of higher-order separation logic into the conjunction rule proving does not affect the proof idea, so we omit the details in this thesis.

We use $e\Downarrow$ to say that the evaluation of e is valid, and $e \Downarrow v$ to say that the evaluation of e is valid and the result of the evaluation is value v. $\&e \Downarrow p$ is used to denote that evaluating e results in a valid addressable variable at the address p in the heap.

In fact, the VST formalization of the abstract Clight language discriminates between the memory load and the memory write operation (both of which are denoted as $e_1 := e_2$ in the text) syntactically with two different Inductive type constructors Sset and Sassign.

For the Sset constructor, which indicates assigning a value to a non-addressable variable, there are two corresponding rules, namely Semax-Set and Semax-Load. The following inversion rule is derivable.

Lemma 4.1 (Inversion on Sset) If $\{P\}\ \mathsf{x} := e\ \left\{Q, [\vec{Q'}]\right\}$, then

$$\begin{aligned} P \vDash\ & \triangleright (e \Downarrow \wedge Q[e/\mathsf{x}]) \vee \\ & \exists \pi\ p\ v.\ \pi \text{ is readable share} \wedge \\ & \quad \triangleright (\&e \Downarrow p \wedge (p \mapsto_\pi v * \mathsf{True}) \wedge Q[v/\mathsf{x}]) \end{aligned}$$

The two disjunction components on the RHS correspond to Semax-Set and Semax-Load respectively, depending on whether the value of e is addressable or not. The two cases are disjoint from each other. Therefore, when we do induction on the statement c, we only need to consider the case where both premises use the same logical rule.

For the Semax-Set case, the conjunction rule holds directly since the variable substitution $[e/\mathsf{x}]$ is the same for both premises.

For the Semax-Load case, issue arises that the permission shares that the two premises use may not be the same, neither are the values referenced by e. The intuition that "if a location can be described by two mapsto predicates with readable permission simultaneously, then the values of the mapsto predicates are the same" can only be justified if we look into the semantic models.

We use $\oplus$ to denote the join relations in separation algebra. The semantic model (known as resource map) of VST, and the permission shares are two instances of the separation algebra. Due to the existence of ghost states, the join relation on resource map does not enjoy some properties of the join relation on permission shares, such as the cross split property and the cancellative property.

Proposition 4.1 (Cross Split Property) A join relation for permission-shares has the cross split property if $\pi = \pi_a \oplus \pi_b \wedge \pi = \pi_c \oplus \pi_d \Rightarrow \exists \pi_{ac}\ \pi_{ad}\ \pi_{bc}\ \pi_{bd}, \pi_{ac} \oplus \pi_{ad} = \pi_a \wedge \pi_{bc} \oplus \pi_{bd} = \pi_b \wedge \pi_{ac} \oplus \pi_{bc} = \pi_c \wedge \pi_{ad} \oplus \pi_{bd} = \pi_d$.

Proposition 4.2 (Cancellative Property) A join relation for permission-shares has the cancellative property if $\pi = \pi_a \oplus \pi_b \wedge \pi = \pi_{a'} \oplus \pi_b \Rightarrow \pi_a = \pi_{a'}$.

Therefore, proofs for memory loading/writing will look into the location of the value being loaded/stored on the resource map, so that we can make use of properties of the join relation on permission shares. We use $r@l$ to refer to the resource on location l of the model r.

Following the idea above, we managed to prove the following lemma that derives the conjunction rule for SEMAX-LOAD.

Theorem 4.3 If π_1 and π_2 are readable shares, then

$$\begin{aligned} & (p \mapsto_{\pi_1} v_1 * \mathsf{True}) \wedge (p \mapsto_{\pi_2} v_2 * \mathsf{True}) \\ \vDash \quad & v_1 = v_2 \wedge p \mapsto_{\pi_1 \cup \pi_2} v_1 * \mathsf{True} \end{aligned}$$

Proof Given a model r that satisfy the LHS of the theorem, by the semantics of $*$, there are two ways to disjointly split this model, say $r = r_1 \oplus r_1'$ and $r = r_2 \oplus r_2'$, where $r_1 \vDash p \mapsto_{\pi_1} v_1$ and $r_2 \vDash p \mapsto_{\pi_2} v_2$.

To show $v_1 = v_2$, it suffices to show that the resources on location l referenced by p in r_1 and r_2 are equal. This is done by inversion on the join relation since both r_1 and r_2 are part of the same model r.

As for the second part, we can pointwisely define the two models r_0, r_0' that constitute $p \mapsto_{\pi_1 \cup \pi_2} v_1 * \mathsf{True}$.

For locations l referenced by p, let the share of $r@l$ be π. From the cross split property of permission shares, we have $(\pi_1 \cup \pi_2) \oplus (\pi \cap (\neg(\pi_1 \cup \pi_2))) = \pi$. We can define $r_0@l$ and $r_0'@l$ to hold the two sub shares as above respectively. For other locations, we can simply take $r_0@l$ and $r_0'@l$ to be $r_1@l$ and $r_1'@l$. It follows that $r_0 \oplus r_0' = r$ and $r_0 \vDash p \mapsto_{\pi_1 \cup \pi_2} v_1$.

4.4 Preciseness of memory write

Sassign constructor corresponds to the SEMAX-STORE rule, which first loads the value of e_2 from the memory and then assigns it to e_1. The SEMAX-ASSIGN rule is formalized as a backward rule using magic wand $-\!\!*$. Before executing the statement, the model should be split into two parts. One part is the model that satisfies the $p \mapsto_\pi v$ predicate. The other part should satisfy the post-condition P when joined with a model where the new value v' is assigned. The inversion lemma for memory writing is as follows:

Lemma 4.2 (Inversion on Sassign) If $\{P\}\, e_1 := e_2 \left\{Q, [\vec{Q'}]\right\}$, then

$$\begin{aligned} P \vDash \quad & \exists \pi\ p\ v\ v'.\ \pi \text{ is writable share} \wedge \\ & \rhd \left(\begin{array}{l} \&e_1 \Downarrow p \wedge e_2 \Downarrow v' \wedge \\ (p \mapsto_\pi v * (p \mapsto_\pi v' -\!\!* Q)) \end{array} \right) \end{aligned}$$

After applying Lemma 4.2 on premises of the conjunction rule, the proof obligation left is as follows, which also requires model-level proving.

Theorem 4.4 If π_1 and π_2 are writable shares, then

$$\begin{aligned} & (p \mapsto_{\pi_1} v_1 * (p \mapsto_{\pi_1} v' -\!\!* P_1)) \\ & \wedge (p \mapsto_{\pi_2} v_2 * (p \mapsto_{\pi_2} v' -\!\!* P_2)) \\ \vDash \quad & v_1 = v_2 \wedge p \mapsto_{\pi_1 \cup \pi_2} v_1 * (p \mapsto_{\pi_1 \cup \pi_2} v' -\!\!* P_1 \wedge P_2) \end{aligned}$$

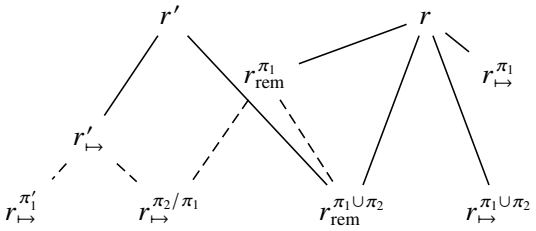

Figure 4–1 Semantic model layout for the proof of Theorem 4.4. (Dotted lines indicate the join relation to be proved)

Proof For any model r that satisfies the LHS of the derivation, there are two ways to disjointly split this model, say $r = r^{\pi_1}_{\mapsto} \oplus r^{\pi_1}_{\text{rem}}$ and $r = r^{\pi_2}_{\mapsto} \oplus r^{\pi_2}_{\text{rem}}$, where $r^{\pi_1}_{\mapsto} \models p \mapsto_{\pi_1} v_1$ and $r^{\pi_2}_{\mapsto} \models p \mapsto_{\pi_2} v_2$.

Applying Theorem 4.3, we can show that $v_1 = v_2$ and that there exists a splitting for $r = r^{\pi_1\cup\pi_2}_{\mapsto} \oplus r^{\pi_1\cup\pi_2}_{\text{rem}}$ where $r^{\pi_1\cup\pi_2}_{\mapsto} \models p \mapsto_{\pi_1\cup\pi_2} v_1$.

We are left to prove that given any model $r'_{\mapsto}$ that satisfies $p \mapsto_{\pi_1\cup\pi_2} v_1$, $r'_{\mapsto} \oplus r^{\pi_1\cup\pi_2}_{\text{rem}}$ satisfies $P_1 \wedge P_2$. Without loss of generality, we show that $r'_{\mapsto} \oplus r^{\pi_1\cup\pi_2}_{\text{rem}} = r' \models P_1$. Figure 4–1 plots the layout of the join relations that have been introduced so far in the proof.

To prove a model satisfying P_1, we must make use of the fact that $r^{\pi_1}_{\text{rem}} \models p \mapsto_{\pi_1} v' \mathrel{-\!\!*} P_1$ The only way to relate r' with $r^{\pi_1}_{\text{rem}}$ is through $r^{\pi_1\cup\pi_2}_{\text{rem}}$, the common sub-model of r' and r. The idea is to "borrow" a sub-model from $r'_{\mapsto}$, to supplement the model $r^{\pi_1\cup\pi_2}_{\text{rem}}$, so that the supplement can be matched with $r^{\pi_1}_{\text{rem}}$.

Based on this observation, we construct two models, which are defined as

$$r^{\pi'_1}_{\mapsto}@l = \begin{cases} [r'_{\mapsto}@l/\pi_1] & \text{if } l \text{ referenced by } p \\ r'_{\mapsto}@l & \text{otherwise} \end{cases}$$

$$r^{\pi_2/\pi_1}_{\mapsto}@l = \begin{cases} [r'_{\mapsto}@l/(\pi_2/\pi_1)] & \text{if } l \text{ referenced by } p \\ \bot & \text{otherwise} \end{cases}$$

, where $[r@l/\pi]$ indicates a resource on location l that has the same value as $r@l$ but with the permissions reassigned as π. The two dotted join relations in Figure 4–1 can be verified.

1. $r'_{\mapsto} = r^{\pi'_1}_{\mapsto} \oplus r^{\pi_2/\pi_1}_{\mapsto}$ follows directly from the fact that $\pi_1 \cup \pi_2 = \pi_1 \oplus (\pi_2/\pi_1)$.
2. $r^{\pi_1}_{\text{rem}} = r^{\pi_2/\pi_1}_{\mapsto} \oplus r^{\pi_1\cup\pi_2}_{\text{rem}}$ requires inversion on $r = r^{\pi_1}_{\text{rem}} \oplus r^{\pi_1}_{\mapsto}$ and $r = r^{\pi_1\cup\pi_2}_{\text{rem}} \oplus r^{\pi_1\cup\pi_2}_{\mapsto}$.

   For l referenced by p, let the permissions for $r@l$, $r^{\pi_1}_{\text{rem}}@l$ and $r^{\pi_1\cup\pi_2}_{\text{rem}}@l$ be π, π_3 and π_4 respectively. Then, we have $\pi = \pi_3 \oplus \pi_1$ and $\pi = \pi_4 \oplus (\pi_1 \cup \pi_2)$. To prove $r^{\pi_1}_{\text{rem}}@l = r^{\pi_2/\pi_1}_{\mapsto}@l \oplus r^{\pi_1\cup\pi_2}_{\text{rem}}@l$, it suffices to show that $\pi_3 = (\pi_2/\pi_1) \oplus \pi_4$. The cancellative property is used here.

   For l not referenced by p, no resources are defined on $r^{\pi_1}_{\mapsto}$ and $r^{\pi_1\cup\pi_2}_{\mapsto}$, so $r^{\pi_1}_{\text{rem}}$ and $r^{\pi_1\cup\pi_2}_{\text{rem}}$ are the same. Since we define no resources for $r^{\pi_2/\pi_1}_{\mapsto}$, the join relation holds. □

Based on the two join relations discovered above, we can use the associativity property of the join relation to show $r' = r^{\pi_1}_{\text{rem}} \oplus r^{\pi'_1}_{\mapsto}$. Then $r' \models P_1$ follows from the premise that $r^{\pi_1}_{\text{rem}} \models p \mapsto_{\pi_1} v' \mathrel{-\!\!*} P_1$.

4.5 Function call with precise specifications

A function specification is denoted as $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$, where P is a precondition parameterized by a list of formal parameters $\vec{y}$, Q is a postcondition parameterized by the return value r, and types $\vec{A}$ is the type of values that is to be shared between P and Q, so both P and Q will also be abstracted over variables of the shared types A.

The Semax-Call rule is formalized as a backward rule. For the function name f to be called, its specification will first be looked up in the typing context Δ, users may choose to apply the specification directly, or apply another specification that is subsumed by the one in Δ. The first choice is available because the subsumption relation is reflexive. The definition of function subsumption is given below, note that the framing of the specification is also incorporated into the definition.

Definition 4.1 (Function subsumption) A function specification $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$ is subsumed by another function specification $[\vec{A'}]\{\lambda\vec{y}.P'\}\ \{\lambda r.Q'\}$ if

$$\begin{aligned}&\forall\vec{x'} : \vec{A'}.\ \forall\vec{b}.\ P'\ \vec{b}\ \vec{x'}\\ &\quad\vDash \exists\vec{x} : \vec{A}.\ \exists F.\ F * P\ \vec{b}\ \vec{x}\wedge\\ &\qquad(\forall a.\ F * Q\ a\ \vec{x}\vDash\ Q'\ a\ \vec{x'})\end{aligned}$$

The body of the rule states that, before executing the statement, the model should be split into two parts, a part that satisfies the specification's pre-condition, and a part that satisfies the call statement's post-condition when joined with a model that satisfies the specification's post-condition.

The Semax-Call rule we use in our system is almost the same as what the current VST release defines, except that we strengthen the pre-condition by moving the existential variables shared by the pre-/post-condition of the function specification to be called into the $\triangleright$ operator. We note such a change is necessary to prove the conjunction rule for the call statement. As a comparison, the pre-condition in the VST version looks like the follows:

$$\left\{\exists\vec{x} : \vec{A}.\ \triangleright\ (\vec{e} \Downarrow \vec{b} \wedge P\,\vec{b}\,\vec{x} * (Q\,\mathsf{a}\,\vec{x} \mathrel{-\!\!*} R))\right\}$$

Besides, we also enforce that the function specification to be called is "precise", defined as follows.

Definition 4.2 (Precise function specification) A function specification $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$ is precise if for any formal parameters $\vec{b}$, return value a and assertions R_1, R_2, it holds that

$$\begin{aligned}&(\exists\vec{x_1} : \vec{A}.\ P\ \vec{b}\ \vec{x_1} * (Q\ a\ \vec{x_1} \mathrel{-\!\!*} R_1))\\ &\wedge(\exists\vec{x_2} : \vec{A}.\ P\ \vec{b}\ \vec{x_2} * (Q\ a\ \vec{x_2} \mathrel{-\!\!*} R_2))\\ \vDash\quad&\exists\vec{x} : \vec{A}.\ P\ \vec{b}\ \vec{x} * (Q\ a\ \vec{x_2} \mathrel{-\!\!*} R_1 \wedge R_2)\end{aligned}$$

The definition looks very similar to Theorem 4–1 that we have proved for the memory writing operation, which can serve as a justification for our definition.

Note that the notion of "precise" we propose is defined with respect to an operation (either an operation on the memory or a function call), while traditionally, "precise" is defined for a predicate in the assertion. A typical example of precise predicates is the $p \mapsto v$ predicate we have seen before.

Definition 4.3 (Precise predicate) A predicate P is precise if for any Q_1, Q_2,

$$(P * Q_1) \wedge (P * Q_2) = P * (Q_1 \wedge Q_2)$$

Clearly, if we remove the existential quantifier $\vec{A}$ from Definition 4.2, then all function specifications written with precise predicates are precise. Therefore, we consider our definition of precise function specification fits into a more general setting, allowing the instantiations of the function specification to be different, while still being able to derive the conjunction rule of function calls, as we will see below.

The proof of conjunction rule for function calls also begins with inversion on premises.

Lemma 4.3 (Inversion on function call) If $\{S\}\,\mathsf{a} := f(\vec{e})\ \left\{R, [\vec{R'}]\right\}$, then

$$\begin{array}{rl} S \vDash & \exists A\ (\lambda\vec{y}.P)\ (\lambda r.Q)\ \vec{b}. \\ & \Delta(f) <: [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} \wedge \\ & \Delta(f) \text{ is a precise specification} \wedge \\ & \rhd \left(\begin{array}{l} \vec{e} \Downarrow \vec{b} \wedge \\ \exists \vec{x} : \vec{A}.\ P\ \vec{b}\ \vec{x} * (Q\ \mathsf{a}\ \vec{x} \mathrel{-\!\!*} R) \end{array} \right) \end{array}$$

By applying Lemma 4.3, the proof obligation is left as follows.

Theorem 4.5 If $\Delta(f) = [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$, $\Delta(f)$ is a precise function specification and

$$\begin{array}{l} [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} <: [\vec{A_1}]\{\lambda\vec{y}.P_1\}\ \{\lambda r.Q_1\} \\ [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} <: [\vec{A_2}]\{\lambda\vec{y}.P_2\}\ \{\lambda r.Q_2\} \end{array}$$

then

$$\begin{array}{rl} & (\exists \vec{x_1} : \vec{A_1}.\ P_1\ \vec{b}\ \vec{x_1} * (Q_1\ a\ \vec{x_1} \mathrel{-\!\!*} R_1)) \\ & \wedge (\exists \vec{x_2} : \vec{A_2}.\ P_2\ \vec{b}\ \vec{x_2} * (Q_2\ a\ \vec{x_2} \mathrel{-\!\!*} R_2)) \\ \vDash & \exists \vec{x} : \vec{A}.\ P\ \vec{b}\ \vec{x} * (Q\ a\ \vec{x_2} \mathrel{-\!\!*} R_1 \wedge R_2) \end{array}$$

To apply the condition that $\Delta(f)$ is a precise function specification, we need to rewrite the assertion with the subsumption relation.

Lemma 4.4 If $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} <: [\vec{A'}]\{\lambda\vec{y}.P'\}\ \{\lambda r.Q'\}$, then for any formal parameters $\vec{b}$, return value a and assertion R,

$$\exists \vec{x'} : \vec{A'}.\ P'\ \vec{b}\ \vec{x'} * (Q'\ a\ \vec{x'} \mathrel{-\!\!*} R) \vDash \exists \vec{x} : \vec{A}.\ P\ \vec{b}\ \vec{x} * (Q\ a\ \vec{x} \mathrel{-\!\!*} R)$$

Proof By logical derivation,

$$\begin{array}{rl} & P'\ \vec{b}\ \vec{x'} * (Q'\ a\ \vec{x'} \mathrel{-\!\!*} R) \\ \vDash & (\exists \vec{x} : \vec{A}.\ \exists F.\ F * P\ \vec{b}\ \vec{x} \wedge (\forall a.\ F * Q\ a\ \vec{x} \vDash\ Q'\ a\ \vec{x'})) \\ & *(Q'\ a\ \vec{x'} \mathrel{-\!\!*} R) \\ \vDash & \exists \vec{x} : \vec{A}.\ \exists F.\ (\forall a.\ F * Q\ a\ \vec{x} \vDash\ Q'\ a\ \vec{x'}) \wedge \\ & F * P\ \vec{b}\ \vec{x} * (Q'\ a\ \vec{x'} \mathrel{-\!\!*} R) \\ \vDash & \exists \vec{x} : \vec{A}.\ \exists F.\ (\ F * Q\ a\ \vec{x} \vDash\ Q'\ a\ \vec{x'}) \wedge \\ & F * P\ \vec{b}\ \vec{x} * (Q'\ a\ \vec{x'} \mathrel{-\!\!*} R) \\ \vDash & \exists \vec{x} : \vec{A}.\ \exists F.\ F * P\ \vec{b}\ \vec{x} * (F * Q\ a\ \vec{x} \mathrel{-\!\!*} R) \\ \vDash & \exists \vec{x} : \vec{A}.\ P\ \vec{b}\ \vec{x} * (Q\ a\ \vec{x} \mathrel{-\!\!*} R) \end{array}$$

Above, we have completed all the conjunction rule proof for the stronger program logic we have defined for VST-A.

4.6 Summary

In this chapter, we first explain why the conjunction rule is essential to VST-A framework in §4.1. Our goal is to derive the conjunction rule from a complex but practical logic like VST. The conjunction rule is proved by induction on the abstract syntax of the program statement. We present a proof draft in §4.2, which handles the cases where structural Hoare logic rules are applied. Next, we show how the conjunction rule is derivable for memory load (§4.3), memory write (§4.4) operations in VST. To prove conjunction rule for function call, we require the function specification to be called to be precise. The preciseness of function specifications is a new notion proposed in this thesis. With precise specifications, the function call is proven to hold the conjunction rule in §4.5.

Chapter 5 Implementation

5.1 Frontend design

In this section, we show the design of VST-A's frontend①, which parses a commented C program and generates annotated program into an abstract called ClightA in Coq. The concrete syntax of ClightA has been described in detail in §3.1. The frontend program is implemented by applying minimal modifications on CompCert Clightgen, which generates Clight program that user wants to verify. So this method allows us to easily guarantee the annotated program is corresponding to original program.

Figure 5–1 shows structure of VST-A's frontend comparing with CompCert's Clight generator (clightgen). Clightgen first parses C code and then does a series of transformations to Clight. Our frontend first includes comments as strings in parsing and syntax transformation. We slightly modify Clightgen so that these comments are all preserved. §5.1.1 explains this part in detail. Then our frontend analyzes the comments and build annotated programs with with annotations in OCaml strings. And finally, our frontend prints the annotated program to a Coq's ".v" file, where the annotated program is a formal Coq term. §5.1.2 explains this part in detail.

5.1.1 Parsing and transformation

Clightgen's parser is built by a verified parser generator[29] and a context free grammar for C. We slightly modify the context free grammar to include comments when parsing. Figure 5–2 shows a comparison between original and modified grammar. LBRACE and RBRACE stands for left brace and right brace in C code to parse. A question mark means this subpattern is optional. In original parser grammar, one kind of statements is compound statement, i.e. a code block enclosed by a pair of braces. There are also other kinds of statements e.g. if-statement and loop. We omit full definition of statement here because it is complex and not relevant to including comment when parsing. Inside a code block, there is a list of statements, which can be empty. In this list, comments may occur on the left end, right end and between two statements. In other words, there is an optional list of comments on the left of all statements and on the right of each statement. So we add an optional list of comments on the left of the list of statements in grammar rule of compound_statement, and we add an optional list of comments to occur on the right of each statement in grammar rule of block_item_list. We believe that this design is robust enough to cope with potential future updates of CompCert.

After parsing, Clightgen applies a series of transformations to convert C to Clight. We modify the intermediate representations to preserve comments in this process. Figure 5–3 shows the comparison between syntax of original Clight and our Clight-with-comment. Two new syntactic combinators are added to represent comments on the left and right side of a statement. We also apply similar changes to other intermediate representations and adapt each transformation step for

① We base our presentation on top of an legacy version of VST-A. The legacy version VST-A[28] is a tactic-based annotation verifier which has a very different backend design from what we have presented in this thesis. The development of the thesis reuses the frontend of VST-A a lot, with slight modifications to the target language.

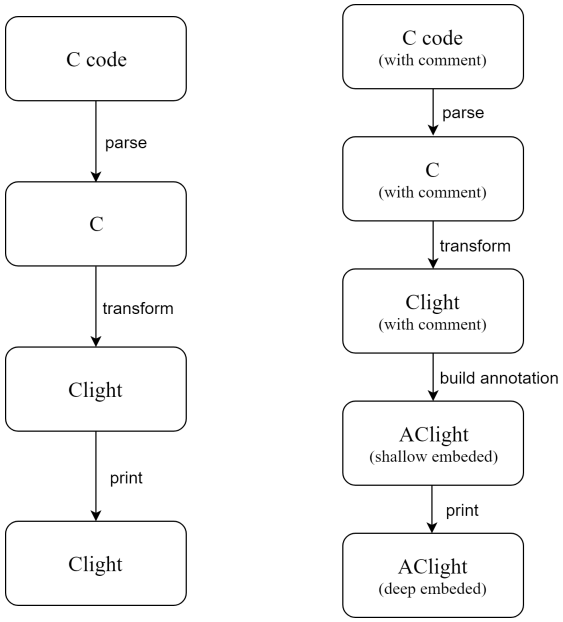

Figure 5–1 Frontend structure

```
statement :=
| compound_statement
| expression_statement
| ...
compound_statement :=
| LBRACE block_item_list? RBRACE
block_item_list :=
| block_item_list? block_item
block_item :=
| statement
```

a) Original context free grammar

```
statement :=
| compound_statement
| expression_statement
| ...
compound_statement :=
| LBRACE comment_list?
      block_item_list? RBRACE
block_item_list :=
| block_item_list? block_item comment_list?
block_item :=
| statement
```

b) Context free grammar for comment

Figure 5–2 Context free grammars comparison

$$\begin{array}{lll} \text{statement}: & c := & c_1; c_2 \\ & | & \text{if } (e)\ c_1 \text{ else } c_2 \\ & | & \text{loop } (c_2)\ c_1 \\ & | & \cdots \end{array}$$

a) Clight

$$\begin{array}{lll} \text{statement}: & c := & /\!*comment*/\!|\ c \\ & | & c\ |\!/*comment*/\!| \\ & | & c_1; c_2 \\ & | & \text{if } (e)\ c_1 \text{ else } c_2 \\ & | & \text{loop } (c_2)\ c_1 \\ & | & \cdots \end{array}$$

b) Clight with comment

Figure 5–3 Syntax of Clight and Clight with comment

preserving comments. Our modifications are all local and small-scale. Also, trivially eliminating comments can turn a Clight-with-comment program to the original Clight program.

5.1.2 From comment to annotation

After getting program in Clight-with-comments, an annotation building process interprets the comments and convert it into annotations. The job of this process has four aspects. We show them in next four paragraphs. After that, we give an example to demonstrate how the annotated program user written is processed. And finally, we show how string annotation is converted to syntactic annotation in the last paragraph.

Reassociating sequential compositions Clight-A requests the left hand side of every sequential composition to be a single statement. This it to guarantee each ex-given clause encloses all statements after it. Annotation building process puts statements in a sequential composition into a list, and fold them from right to left.

Finding loop invariants Loop invariants can appear before loops as comments, but they are not component of loop in syntax of Clight-A. Also, loop invariants may not occur right before loops, because the initializing statements in for loops are transformed into separated statements before loops, and these statements will appear between loops and loop invariants. So annotation builder finds the closest loop invariants that users mark as /* Inv ... */ in the comments, and insert them into the correct position as assertion constructors in the Clight-A program.

Our annotation builder allows users to write zero, one or two loop invariants. The loop without loop invariant annotations does not need special treatment in parsing, but it can only be split successfully if the assertions in the loop cover all control flows. Otherwise an error will be raised by the split function. Users are allowed to write only one loop invariant instead of two for $\mathsf{loop}\ (c_{\text{incr}})\ c$. bacause in most scenarios, c_{incr} is skip, or there are no continue statements in c (excluding those in inner loops). In the first case, annotation builder translates the single invariant directly into an assertion at the beginning of c, because $\{I\}\ \mathsf{skip}\ \{I\}$ is always provable. In the second case, annotation builder transforms the loop into $\mathsf{loop}\ (\mathsf{skip})\ \{c; c_{\text{incr}}\}$ and then translates the single invariant like the first case. This transformation is proved to be sound in VST, and VST also uses this transformation for such kind of loops.

Generating given clauses VST suggests proving Hoare triples when precondition is in canonical form without existential variables. Our VST-A framework shares the same idea by making existential variables explicit in the abstract syntax ClightA and split results. In order to parse an annotated C program, every assertion with existential variables will generate an ex-given clause in the target abstract syntax. For users' convenience, annotation builder finds these existential variables in assertions and generates given clauses after them automatically. Annotation builder also finds existential variables in loop invariants and generates given clauses at the beginning of loop bodies and incremental statements in the same way. These automatically generated given clauses cover all the cases that given clauses are needed, so users do not need to write any given clauses manually.

```
1  struct list {unsigned head; struct list *tail;};
2  struct list *reverse (struct list *p) {
3    // With  l,
4    // Require  ll([[p]], l)
5    // Ensure  ll([[ret]], rev(l))
6    struct list *w, *t, *v;
7    w = NULL; v = p;
8    // Inv  ∃ a b l1 l2. [[w]] = a ∧ [[v]] = b ∧ l = rev(l1) l2 ∧ ll(a, l1) ∗ ll(b, l2)
9    while (v) {
10     // Assert  ∃ c x l2'. [[w]] = a ∧ [[v]] = b ∧ l2 = x l2' ∧ b ↦ (x, c) ∗ ll(a, l1) ∗ ll(c, l2')
11     t = v->tail; v->tail = w; w = v; v = t;
12   }
13   return w;
14 }
```

a) The annotated program that user writes

```
1  struct list {unsigned head; struct list *tail;};
2  reverse_spec :=
3    (λ l.  (ll([[p]], l), ll([[ret]], rev(l))))
4  reverse_hint :=
5    // Given  l,
6    w = NULL; v = p;
7    loop (skip) {
8      // ExGiven  a b l1 l2,
9      // Assert  [[w]] = a ∧ [[v]] = b ∧ l = rev(l1) l2 ∧ ll(a, l1) ∗ ll(b, l2)
10     if (v)
11       skip;
12     else
13       break;
14     // ExGiven  c x l2'.
15     // Assert  [[w]] = a ∧ [[v]] = b ∧ l2 = x l2' ∧ b ↦ (x, c) ∗ ll(a, l1) ∗ ll(c, l2')
16     t = v->tail; v->tail = w; w = v; v = t;
17   }
18   return w;
```

b) The annotated program that annotation builder generates

Figure 5–4 Example: frontend's effects

Extracting function specifications We allow users to write function specification at the beginning of each function using With, Require and Ensure keywords as shown in examples, e.g. , This is an easy and clear when to write function specifications. Annotation builder extracts function specifications by finding With-Require-Ensure pattern at the beginning of each function and print it as a function of multiple parameters. Later a Coq tactic will convert this function to a formal function specification that conforms Verifiable C logic in VST. Assertions may also use variables mentioned in with clause. So annotation builder also generate given clauses that effect on the whole function body. These given clauses at the beginning of function body are bound to variables in with clause when initializing the annotation based verifier, such that they will not be not a part of the annotated program in the later splitting process.

$$
\begin{array}{rrl}
\text{string}: & s := & \cdots \\
\text{statement}: & c := & \text{ExGiven } s_x\ s_P\ c \\
& \mid & \text{Assert } s \\
& \mid & \mathsf{loop}\ (c_2)\ c_1 \\
& \mid & \cdots
\end{array}
$$

a) Syntax of Clight-A with string annotation

$$
\begin{array}{rrl}
\text{assertion}: & P := & \cdots \\
\text{identifier}: & x := & \cdots \\
\text{type}: & A := & \cdots \\
\text{statement}: & c := & \text{ExGiven } x : A,\ \{P(x)\}\ C \\
& \mid & \text{Assert } P \\
& \mid & \mathsf{loop}\ (c_2)\ c_1 \\
& \mid & \cdots
\end{array}
$$

b) Syntax of Clight-A with syntactic annotation

Figure 5–5 Comparison between string annotation and syntactic annotation

Example Figure 5–4 shows an example. First, frontend parses annotated program written by user with comments and preforms transformations to Clight with comment, e.g. replacing while-loop with general loop. Then, function specification is extracted from comment to a function of l. User only gives one loop invariant. It is translated into assertions in the loop by annotation builder. The ex-given clause is also generated automatically. User does not write an assertion after the loop because annotation builder checks that the loop has only one normal exit point, so it allows user to omit this assertion.

Printing It is worth mentioning that annotation builder generates Clight-A with string annotation, instead of Clight-A with syntactic annotation. Figure 5–5 shows the differences between string annotation and syntactic annotation. Strings are more flexible and easier to process. When printing to Coq's ".v" file, string annotations are printed at proper positions and become syntactic annotation.

5.2 Coq formalization

All of the definitions and proofs in this thesis have been formalized in Coq (8.10) proof assistant, with dependency on VST 2.5.

There are three major components in the artifact:

- The `frontend` directory contains the Coq files that describes the abstract syntax of the intermediate languages our frontend parser uses. Some frontend processing functions are also implemented in Coq and then extracted as OCaml programs. There are some OCaml programs that link the extracted functions together with I/O and command line arguments. The files in `frontend` will together compile to a single executable `aclightgen` that takes an assertion annotated `.c` file and outputs a `.v` file of its ClightA abstract syntax in Coq.
- The `CSplit` directory contains all the core definitions and proofs of our split function. We have listed the details in Table 5–1.
- We choose to use VST as the verification backend. In order to fit our newly defined stronger variant of Verifiable C logic, we migrate some of VST's automated proof tactics in the `floyd-seq` directory, so that users can use them when they verify the correctness of their

path splitting results.

Table 5–1 Coq file summary for the VST-A control flow splitting core (in `CSplit` directory)

Coq File Name	Contents	Lines of Code
vst_ext.v	auxiliary VST definitions	193
model_lemmas.v	new VST model level lemmas	1736
logic_lemmas.v	new VST logical level lemmas	197
strong.v	the stronger variant of Verifiable C logic, and the proof of its conjunction rule	2534
AClight.v	definitions of AClight syntax, split result interface and split function	2278
semantics.v	interpretation of split results, and lemmas about them	2305
soundness.v	soundness theorem of split function	1516
AClightFunc.v	an equivalent split function for AClight.v adjusted for better split result printing	1962
total		13907

5.3 Backend design

We have presented a general workflow of verifying programs in VST-A in Figure 1–2. Now we use Figure 5–6 to refine the workflow with a concrete example of verifying the `reverse` program we have presented in Figure 1–1 to see what files VST-A backend generates.

Our concrete implementation is based on a single `Makefile` that combines the process of parsing, path splitting and automatic verification (if any).

Users are required to write programs and annotations in `.c` files. If the annotation requires extra predicates, such as `ll` in the `reverse` example, then they also need to write an auxiliary Coq file to define them. Next, VST-A will automatically parse the source program, and call the split function we have implemented in Coq to print the split results into separate `.v` files.

Users are left to prove the correctness of the split results. The proof split results may have dependency on `reverse_def.v` and `reverse_prog.v` since assertions may use auxiliary predicates and information of program variables (such as their names and C types) are defined in the `clightgen` results.

After all the paths have been verified, an automatically generated `reverse_verif.v` will import the path verification results, and then establish the correctness of the program. The `reverse_verif.v` script will provide a theorem of the Hoare triple of the whole program, in the same format as VST, but proved in a different way by using our soundness theorem and users' path verification results.

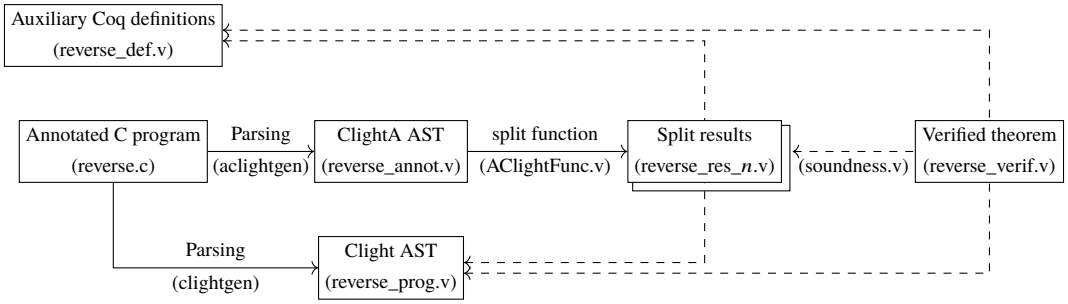

Figure 5–6 Detailed workflow in VST-A (dotted arrows for Coq dependency)

5.4 Summary

In this chapter, based on the theories that have been built up in the previous chapters, we demonstrate how the VST-A framework is implemented. We present the frontend design in §5.1. Then we evaluate the Coq formalization of the VST-A framework in §5.2. Finally, we show how a concrete program is verified by the VST-A framework in §5.3 to demonstrate the backend design.

Chapter 6 Conclusion

This thesis presents a new framework, VST-A, for verification of assertion annotated programs. The framework is based on the idea of splitting the verification of a program into the verification of several control flow paths seprated by assertions. I define a formal language Clight-A for assertion annotated programs as the input language of the framework. The core of VST-A is an assertion-based control flow splitting algorithm that splits the original Hoare triple into straightline Hoare triples of control flow paths. I design the algorithm, implement the split function in Coq and formally prove its soundness.

The conjunction rule is essential for the soundness of the framework. To make it available in VST-A, I define a stronger variant of VST's program logic, which removes the logical operators related to concurrency and requires function specifications being called during verification to be precise. I formalize the logic in Coq and prove that the conjunction rule holds for the stronger logical system, thus ensuring foundational soundness of the VST-A framework.

Based on the theoritical results above, I implement a prototype system that demonstrates the whole workflow of the framework. I modify the frontend of the original VST-A framework to build Clight-A abstract syntax trees from C programs with assertion annotations as comments. I set up an automated workflow to call the split function and generate path splitting results into separate Coq files for users to prove. To improve usability, I migrate proof automation tactics from VST to the stronger logical system defined in VST-A. My Coq formalization of the framework totals about 14,000 lines of code. The resulting system can be used to verify various useful C programs such as mathematical functions, linked-list manipulation, binary search tree operations, and so on, as demonstrated by examples.

In summary, VST-A is a foundationally sound verification tool for real-world C programs. It enjoys a rich assertion language and flexible proof strategies, and meanwhile allows users to write readable assertion annotations directly as comments.

Based on achievements in this thesis, more future work of VST-A can be done. First, the current VST-A system focuses on verification of complex sequential programs, the requirement of the conjunction rule makes it non-trivial to extend to concurrent program verification. It is interesting to study whether the conjunction rule can be proved with more delicate restrictions such as requiring the ghost state to be empty. Besides, our prototype implementation does not consider the efficiency of the framework as a major concern. When the annotations involves too many existential varibles, the split function may be slow to run due to our shallow embedding treatment of logical variables in Coq. A more efficient implementation is possible to make the framework more powerful in verifying large scale programs. Moreover, the new notion of precise function specifications, which is proposed for the program logic to derive the conjunction rule, is also worth exploring, since it allows more properties than the tranditional notion of precise predicates does. In the current framework, users need to prove that a function specification is precise when they call it during verification. Common patterns of precise function specifications can be summarized and built into VST-A to further relieve users' verification burden.

Bibliography

[1] GUÉNEAU A, MYREEN M O, KUMAR R, et al. Verified characteristic formulae for CakeML[C/OL]//Lecture Notes in Computer Science: vol. 10201 LNCS. Springer Verlag, 2017: 584-610. https://link.springer.com/chapter/10.1007/978-3-662-54434-1_22. DOI: 10.1007/978-3-662-54434-1_22.

[2] KREBBERS R, TIMANY A, BIRKEDAL L. Interactive proofs in higher-order concurrent separation logic[J/OL]. ACM SIGPLAN Notices, 2017, 52(1): 205-217. http://dx.doi.org/10.1145/3009837.3009855. DOI: 10.1145/3009837.3009855.

[3] JUNG R A, KREBBERS R, JOURDAN J H, et al. Iris from the ground up: A modular foundation for higher-order concurrent separation logic[J/OL]. Journal of Functional Programming, 2018, 28: 73. https://www.cambridge.org/core/journals/journal-of-functional-programming/article/iris-from-the-ground-up-a-modular-foundation-for-higherorder-concurrent-separation-logic/26301B518CE2C52796BFA12B8BAB5B5F. DOI: 10.1017/S0956796818000151.

[4] APPEL A W. Verified software toolchain (Invited talk)[C/OL]//Lecture Notes in Computer Science: vol. 6602 LNCS. Springer, Berlin, Heidelberg, 2011: 1-17. https://link.springer.com/chapter/10.1007/978-3-642-19718-5_1. DOI: 10.1007/978-3-642-19718-5_1.

[5] CAO Q, BERINGER L, GRUETTER S, et al. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs[J]. Journal of Automated Reasoning, 2018, 61(1-4): 367-422. DOI: 10.1007/S10817-018-9457-5.

[6] BERINGER L. Verified Software Units[C/OL]//Lecture Notes in Computer Science: vol. 12648 LNCS. Springer, Cham, 2021: 118-147. https://link.springer.com/chapter/10.1007/978-3-030-72019-3_5. DOI: 10.1007/978-3-030-72019-3_5.

[7] BOUTILLIER P, GLONDU S, GRÉGOIRE B, et al. Coq 8.4 Reference Manual[J/OL]., 2014. https://hal.inria.fr/hal-01114602.

[8] PAULSON L C. Isabelle - A Generic Theorem Prover[M]. Springer-Verlag, 1994: 321.

[9] LEINO K R M. Dafny: An automatic program verifier for functional correctness[C]//Lecture Notes in Computer Science: vol. 6355 LNAI. 2010: 348-370. DOI: 10.1007/978-3-642-17511-4_20.

[10] CHIN W N, DAVID C, NGUYEN H H, et al. Automated verification of shape, size and bag properties via user-defined predicates in separation logic[J]. Science of Computer Programming, 2012, 77(9): 1006-1036. DOI: 10.1016/j.scico.2010.07.004.

[11] JACOBS B, SMANS J, PHILIPPAERTS P, et al. VeriFast: A powerful, sound, predictable, fast verifier for C and Java[C/OL]//Lecture Notes in Computer Science: vol. 6617 LNCS. 2011: 41-55. http://www.cs.kuleuven.be/. DOI: 10.1007/978-3-642-20398-5_4.

[12] MÜLLER P, SCHWERHOFF M, SUMMERS A J. Viper: A verification infrastructure for permission-based reasoning[G/OL]//Dependable Software Systems Engineering: vol. 9583. Springer Verlag, 2017: 104-125. https://link.springer.com/chapter/10.1007/978-3-662-49122-5_2. DOI: 10.3233/978-1-61499-810-5-104.

[13] BAUDIN P, BOBOT F, BÜHLER D, et al. The dogged pursuit of bug-free C programs: The Frama-C Software Analysis Platform[EB/OL]. Association for Computing Machinery. 2021. https://cacm.acm.org/magazines/2021/8/254311-the-dogged-pursuit-of-bug-free-c-programs/fulltext.

[14] KROENING D, TAUTSCHNIG M. CBMC - C Bounded Model Checker[C/OL]//Lecture Notes in Computer Science: vol. 8413 LNCS. Springer, Berlin, Heidelberg, 2014: 389-391. https://link.springer.com/chapter/10.1007/978-3-642-54862-8_26. DOI: 10.1007/978-3-642-54862-8_26.

[15] IVANČIĆ F, YANG Z, GANAI M K, et al. F-SOFT: Software verification platform[C/OL]//Lecture Notes in Computer Science: vol. 3576. Springer, Berlin, Heidelberg, 2005: 301-306. https://link.springer.com/chapter/10.1007/11513988_31. DOI: 10.1007/11513988_31.

[16] IVANČIĆ F, BALAKRISHNAN G, GUPTA A, et al. Scalable and scope-bounded software verification in Varvel[J/OL]. Automated Software Engineering, 2015, 22(4): 517-559. https://link.springer.com/article/10.1007/s10515-014-0164-0. DOI: 10.1007/s10515-014-0164-0.

[17] CALCAGNO C, DISTEFANO D. Infer: An automatic program verifier for memory safety of C programs[C/OL]//Lecture Notes in Computer Science: vol. 6617 LNCS. Springer, Berlin, Heidelberg, 2011: 459-465. https://link.springer.com/chapter/10.1007/978-3-642-20398-5_33. DOI: 10.1007/978-3-642-20398-5_33.

[18] REYNOLDS J C. Separation logic: A logic for shared mutable data structures[C]// Proceedings - Symposium on Logic in Computer Science. 2002: 55-74. DOI: 10.1109/lics.2002.1029817.

[19] GOTSMAN A, BERDINE J, COOK B. Precision and the conjunction rule in concurrent separation logic[C]//Electronic Notes in Theoretical Computer Science: vol. 276: 1. 2011: 171-190. DOI: 10.1016/j.entcs.2011.09.021.

[20] VAFEIADIS V. Concurrent separation logic and operational semantics[C]//Electronic Notes in Theoretical Computer Science: vol. 276: 1. Elsevier, 2011: 335-351. DOI: 10.1016/j.entcs.2011.09.029.

[21] BERTOT Y, CASTÉRAN P. Interactive Theorem Proving and Program Development[M/OL]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004: 490. http://link.springer.com/10.1007/978-3-662-07964-5%20http://www.springer.com/us/book/9783540208549. DOI: 10.1007/978-3-662-07964-5.

[22] LEROY X. Formal verification of a realistic compiler[J/OL]. Communications of the ACM, 2009, 52(7): 107-115. https://dl.acm.org/doi/abs/10.1145/1538788.1538814. DOI: 10.1145/1538788.1538814.

[23] HOARE C A. An axiomatic basis for computer programming[J/OL]. Communications of the ACM, 1969, 12(10): 576-580. https://dl.acm.org/doi/abs/10.1145/363235.363259. DOI: 10.1145/363235.363259.

[24] APPEL A W, DOCKINS R, HOBOR A, et al. Program logics for certified compilers[M]. 2014: 1-451. DOI: 10.1017/CBO9781107256552.

[25] BORNAT R, O'HEARN P, CALCAGNO C, et al. Permission accounting in separation logic[C/OL]//ACM SIGPLAN Notices: vol. 40: 1. ACM PUB27 New York, NY, USA,

2005: 259-270. https://dl.acm.org/doi/abs/10.1145/1047659.1040327. DOI: 10.1145/1047659.1040327.

[26] DOCKINS R, HOBOR A, APPEL A W. A fresh look at separation algebras and share accounting[C/OL]//Lecture Notes in Computer Science: vol. 5904 LNCS. 2009: 161-177. http://www.cs.princeton.edu/$%5Csim$appel/sa-accounting/proofs.tar.gz. DOI: 10.1007/978-3-642-10672-9_13.

[27] BERINGER L, APPEL A W. Abstraction and subsumption in modular verification of C programs[J/OL]. Formal Methods in System Design, 2021, 58(1-2): 322-345. https://link.springer.com/article/10.1007/s10703-020-00353-1. DOI: 10.1007/s10703-020-00353-1.

[28] WANG Q, CAO Q. VST-A: A Foundationally Sound Annotation Verifier[J/OL]., 2019. arXiv: 1909.00097. https://arxiv.org/abs/1909.00097v1%20http://arxiv.org/abs/1909.00097.

[29] JOURDAN J H, POTTIER F, LEROY X. Validating LR(1) parsers[C/OL]//Lecture Notes in Computer Science: vol. 7211 LNCS. Springer, Berlin, Heidelberg, 2012: 397-416. https://link.springer.com/chapter/10.1007/978-3-642-28869-2_20. DOI: 10.1007/978-3-642-28869-2_20.