VST-A: A Foundationally Sound Annotation Verifier

LITAO ZHOU, Shanghai Jiao Tong University, China and The University of Hong Kong, China
JIANXING QIN, Shanghai Jiao Tong University, China
QINSHI WANG, Princeton University, United States
ANDREW W. APPEL, Princeton University, United States
QINXIANG CAO[*], Shanghai Jiao Tong University, China

Program verifiers for imperative languages such as C may be *annotation-based*, in which assertions and invariants are put into source files and then checked, or tactic-based, where proof scripts separate from programs are *interactively* developed in a proof assistant such as Coq. Annotation verifiers have been more automated and convenient, but some interactive verifiers have richer assertion languages and formal proofs of soundness. We present VST-A, an annotation verifier that uses the rich assertion language of VST, leverages the formal soundness proof of VST, but allows users to describe functional correctness proofs intuitively by inserting assertions.

VST-A analyzes control flow graphs, decomposes every C function into control flow paths between assertions, and reduces program verification problems into corresponding *straightline Hoare triples*. Compared to existing foundational program verification tools like VST and Iris, in VST-A such decompositions and reductions can nonstructural, which makes VST-A more flexible to use.

VST-A's decomposition and reduction is defined in Coq, proved sound in Coq, and computed call-by-value in Coq. The soundness proof for reduction is totally logical, independent of the complicated semantic model (and soundness proof) of VST's Hoare triple. Because of the rich assertion language, not all reduced proof goals can be automatically checked, but the system allows users to prove residual proof goals using the full power of the Coq proof assistant.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Theory of computation → Program verification**; *Hoare logic*; Automated reasoning; Parsing.

Additional Key Words and Phrases: Annotated Programs, Foundational Verification, Coq

ACM Reference Format:
Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proc. ACM Program. Lang.* 8, POPL, Article ? (January 2024), 30 pages.

1 INTRODUCTION

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

Interactive program verification tools, such as Iris [Jung et al. 2018; Krebbers et al. 2017b] and VST [Appel 2011; Beringer 2021; Cao et al. 2018], are built in interactive theorem provers like Coq or HOL4. Users write formal program correctness proofs in the same theorem prover, by using the lemmas and tactics of the program verification tool.

Some of these interactive 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 programs and higher-level properties such as functional correctness. First,

[*]Corresponding author.

Authors' addresses: Litao Zhou, Shanghai Jiao Tong University, China and The University of Hong Kong, China, tonyzhou0608@gmail.com; Jianxing Qin, Shanghai Jiao Tong University, China, qdelta@sjtu.edu.cn; Qinshi Wang, Princeton University, United States, qinshiw@cs.princeton.edu; Andrew W. Appel, Princeton University, United States, appel@princeton.edu; Qinxiang Cao, Shanghai Jiao Tong University, China, caoqinxiang@gmail.com.

real-world programming languages are complicated. For example, it is very subtle to determine what C programs may cause undefined behavior. Second, an advanced program logic for higher-order properties usually has a nontrivial soundness proof. For example, VST and Iris use step-indexed semantics to interpret impredicative assertion languages whose soundness proof is complicated.

Interactive program verification tools can also benefit from the rich logic language and the rich proof language of theorem provers (like Coq and HOL). These tools can easily shallow-embed higher-order functions and predicates in their assertion languages. They also make it convenient (in specifying programs and program assertions) to introduce additional logical connectives. Additionally, the tactic proof languages in proof assistants are very powerful when users need to describe proof strategies such as proof-by-induction and proof-by-contradiction.

A different strain of program verification tools requires programmers to write annotations in the source code. With sufficient annotations, tools like Dafny [Leino 2010], Hip/Sleek [Chin et al. 2012], VeriFast [Jacobs et al. 2011], Viper [Müller et al. 2017], Frama-C [Baudin et al. 2021] and CN [Pulte et al. 2023] can verify program correctness automatically. By restricting the assertion languages, tools like CBMC [Kroening and Tautschnig 2014], F-Soft [Ivančić et al. 2015, 2005], and Infer [Calcagno and Distefano 2011] can reduce the annotation overhead for programmers while preserving automation.

Compared with writing tactical proofs in a theorem prover, we believe that *writing annotations is a much more straightforward way of demonstrating that a program is correct.* Even proofs in theory papers and proofs written completely in an interactive prover are often presented in research papers as annotated programs, e.g. Reynolds's first paper about separation logic uses annotations to describe separation logic proofs [Reynolds 2002, page 10], and recent verification papers like Jung et al. [2020]'s work extending Iris to support prophercy variables also uses annotations to describe their proofs [Jung et al. 2020, page 7]. Fig. 1 shows an implementation of an in-place linked list reversal and its functional correctness proof.[1] The annotations on lines 3-5 describe the specification this function should satisfy: for any list of integers l (the With clause on line 3), if l is stored in a linked list and this linked list's head pointer is passed to the function by the program variable p (the Require clause on line 4), then the function reverses the linked list and returns the new head pointer (the Ensure clause on line 5). The assertion on line 9 describes the main idea of a functional correctness proof. It states the criteria that the program state should satisfy every time the program enters the loop body. Assertion-annotated programs can present proofs succinctly; by contrast, in interactive verifiers, key insights into program correctness easily get mixed with structural tactics and become lengthy proof scripts.

In this paper, we demonstrate how to combine the benefits of interactive tools and annotation verifiers. We present VST-A, a foundationally sound verification tool, that is implemented on top of VST. VST-A enjoys rich assertion languages and flexible proof strategies, and it allows users to write readable assertion annotations directly as comments exactly as in Fig. 1.

We illustrate the VST-A workflow in Fig. 2: (1) Users first provide a C program with assertion annotations. (2a) Our front-end parser then converts the source code into ClightA, the Coq representation of this annotated C language. (2b) Next, the C program's functional correctness is reduced to smaller proof goals using the annotations. Specifically, a *split function* accepts a ClightA program and its pre-/post-conditions as input, and returns a set of straightline Hoare triples, each of which consists of a sequence of primary statements (assignment statements, function calls) and/or **assume** commands. For example, Fig. 3 shows the split result of the reverse function in

[1]This is a separation logic [Reynolds 2002] 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. $\mathrm{ll}(\mathrm{p}, l)$ is a separation logic predicate that asserts on the location referenced by variable p stores a linked list of l.

```
struct list {unsigned head; struct list *tail;};
struct list *reverse (struct list *p) {
    /*@ With  l,
        Require  ll (p, l)
        Ensure  ll (ret, rev(l)) */
    struct list *w, *t, *v;
    w = NULL; v = p;
    while (v) {
        /*@ Assert ∃ l1 u x l2. l = rev(l1) x l2 ∧  v ↦ (x, u)  ∗  ll (w, l1)  ∗  ll (u, l2) */
         t = v−>tail; v−>tail = w; w = v; v = t; }
    return w; }
```

Fig. 1. Annotations for verifying linked-list reversal

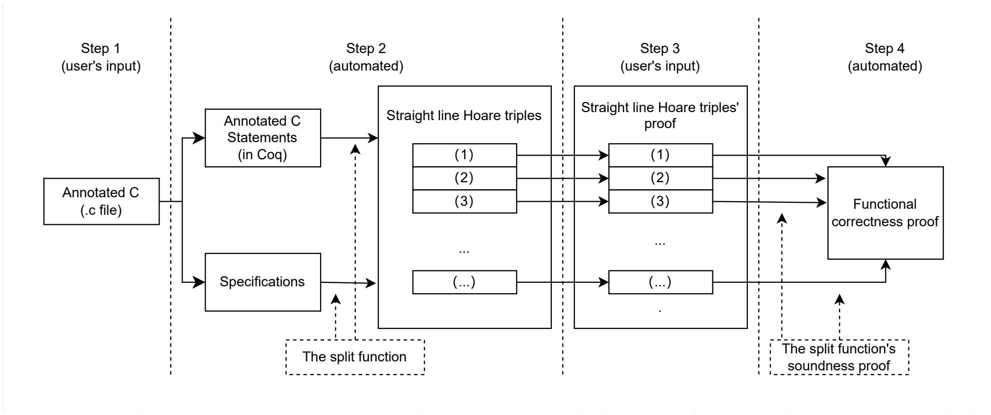

Fig. 2. Verification workflow in VST-A

Fig. 1: four triples are returned as verification goals. As illustrated by the control flow graph in Fig. 4, the functionality of this *split function* is natural; it computes all of the control flow paths that are separated by assertion annotations in the source program. (3) Finally, users are left to prove each straightline Hoare triple in the split result. (4) The VST-A soundness theorem ensures the correctness of the original program if all of the paths have been verified.

In summary, the contributions of this paper are:

(1) **A new framework for program verification** that combines the benefits of interactive provers and the readability of annotated programs.
(2) **A formal language for annotated programs**: We define ClightA, a formal language for annotated C programs, as a method of describing how the functional correctness proof for a large program can be reduced.
(3) **A control-flow-based verification splitting algorithm**: The VST-A proof reduction framework uses a split function that is implemented in Coq and proved sound w.r.t. the VST program logic. We believe this split algorithm and its soundness proof are general and can be applied to other Hoare-style imperative verification tools as well.

(1) $\forall\, l$,

$\{\ \mathrm{ll}\,(\mathrm{p}, l)\ \}$

w = NULL; v = p; **assume** v;

$\left\{\begin{array}{l} \exists\, l_1\ c\ x\ l_2.\ l = \mathrm{rev}(l_1)\ x\ l_2\ \wedge \\ \mathrm{v} \mapsto (x, c)\ *\ \mathrm{ll}\,(\mathrm{w}, l_1)\ *\ \mathrm{ll}\,(c, l_2) \end{array}\right\}$

(2) $\forall\, l$,

$\{\ \mathrm{ll}\,(\mathrm{p}, l)\ \}$

w = NULL; v = p; **assume** !v; ret = w;

$\{\ \mathrm{ll}\,(\mathrm{ret}, \mathrm{rev}(l))\ \}$

(3) $\forall\, l\ l_1\ c\ x\ l_2$,

$\left\{\begin{array}{l} l = \mathrm{rev}(l_1)\ x\ l_2\ \wedge\ \mathrm{v} \mapsto (x, c)\ * \\ \mathrm{ll}\,(\mathrm{w}, l_1)\ *\ \mathrm{ll}\,(c, l_2) \end{array}\right\}$

t = v−>tail; v−>tail = w;

w = v; v = t; **assume** v;

$\left\{\begin{array}{l} \exists\, l_1\ c\ x\ l_2.\ l = \mathrm{rev}(l_1)\ x\ l_2 \wedge \\ \quad \mathrm{v} \mapsto (x, c)\ *\ \mathrm{ll}\,(\mathrm{w}, l_1)\ *\ \mathrm{ll}\,(c, l_2) \end{array}\right\}$

(4) $\forall\, l\ l_1\ c\ x\ l_2$,

$\left\{\begin{array}{l} l = \mathrm{rev}(l_1)\ x\ l_2\ \wedge\ \mathrm{v} \mapsto (x, c)\ * \\ \mathrm{ll}\,(\mathrm{w}, l_1)\ *\ \mathrm{ll}\,(c, l_2) \end{array}\right\}$

t = v−>tail; v−>tail = w;

w = v; v = t; **assume** !v;

ret = w;

$\{\ \mathrm{ll}\,(\mathrm{ret}, \mathrm{rev}(l))\ \}$

Fig. 3. Split results for verifying linked-list reversal

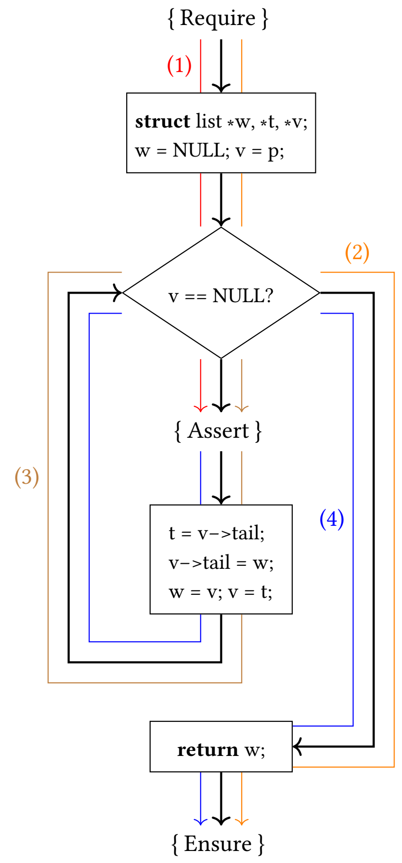

Fig. 4. Control flow graph of linked-list reversal

In the rest of this paper: We will introduce background information about VST-A in §2. We will present our annotation-based proof language and analyze its expressiveness in §3. We will define the split function and prove it sound in §4. We will discuss the connection between our soundness proof and Hoare logic's conjunction rule in §5. We put statistics of VST-A verification examples in §6. Finally, we will discuss related works in §7 and conclude in §8.

2 BACKGROUND

We used Coq [Boutillier et al. 2014] and VST to implement our annotation verifier, VST-A. VST is an interactive program verification tool that is built in Coq. Its primary components are

(1) Verifiable C (§2.2, §2.3), an impredicative higher-order concurrent separation logic that is defined for an abstract C language called Clight (§2.1),
(2) VST-Floyd (§2.4) [Cao et al. 2018], a proof automation system for forward symbolic execution-based verification that efficiently applies VST to real-world C program verification,
(3) A machine-checked soundness proof of Verifiable C in terms of CompCert Clight semantics. Together with the correctness proof of the verified C compiler—CompCert [Leroy 2009], we can obtain the foundational soundness of VST-A w.r.t. the assembly language.

$$\begin{array}{rrl}
\text{expression}: & e := & \cdots \\
\text{primary statement}: & c_p := & e_1 := e_2 \mid e := f(\vec{e}) \\
\text{Clight statement}: & c := & c_p \mid c_1; c_2 \mid \mathsf{if}\ (e)\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{loop}\ (c_2)\ c_1 \\
 & \mid & \mathsf{skip} \mid \mathsf{break} \mid \mathsf{continue} \mid \mathsf{return}
\end{array}$$

Fig. 5. Clight: abstract C language

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

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

Fig. 6. Representative proof rules of C Hoare logic (Part I: compositional rules)

2.1 Clight: abstract C language

We reason about C programs using CompCert Clight's syntax and semantics. Fig. 5 shows a simplified version of its syntax.[2] Clight expressions are side-effect free. CompCert Clight distinguishes assignment statements and function call statements from other statements, and we refer to them as *primary statements*, since they are the basic building blocks for our VST-A development.

CompCert Clight uses loop $(c_{\text{incr}})\ c$ as a general way to describe loops, and it is equivalent to 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 (along with break and continue statements). In the paper presentation, we assume that return statements do not return a value, but our implementation does handle return statements with return values.[3]

2.2 Hoare logic for C programs

VST-A reuses VST's Hoare logic rules, which are known as Verifiable C. VST's Hoare judgment extends the postcondition into four parts to address control flow instructions such as break, continue and return. A judgment

$$\{P\}\ c\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}$$

can be interpreted as starting from a program state that satisfies P. After executing c, if the statement exits normally, the program state satisfies Q. Similarly, the program state should satisfy Q_{brk}, Q_{con}, and Q_{ret} when the statement exits with break, continue, or return, respectively. We use $\left[\vec{Q}\right]$ as an abbreviation of the last three postconditions.

In Verifiable C, most of the compositional rules are standard. Fig. 6 shows some representative ones.[4] To fit the general loop syntax of Clight, the Semax-Loop rule has two invariants, loop invariant I and continue invariant I_{con}. The loop invariant I is required to hold before each iteration,

[2]VST-A does not support goto statements, since they are not supported by VST's program logic. VST-A does not support switch statements for now, but it will be easy to add that to VST-A in the future. VST-A does not yet support CompCert's special calls to built-in functions; these are rarely used in the source language.

[3]VST handles return values by a reserved variable called ret, as we illustrated in Fig. 3. Supporting return statements with return values does not cause significant difficulties in our development.

[4]A full list of rules can be found in the extended version of the paper [?].

and I_{con} is required to hold at continue or before c_{incr} statements. Verifiable C's proof rules for primary statements are less important to end users — VST provides verified forward symbolic execution (§2.4), and thus VST's users do not need to use those rules directly. All proof rules in Verifiable C are proved sound foundationally w.r.t. the CompCert Clight semantics, and symbolic execution applies the proof rules. VST also provides some useful derived rules, a few of which are shown in Fig. 7. The Seq-Assoc rule reassociates sequential compositions, and rules Extract-Exists and Extract-Pure introduce variables and propositions from precondition to context, respectively.

$$\textsc{Extract-Pure}\ \frac{\text{pure}(P_{\text{pure}}) \qquad P_{\text{pure}} \Rightarrow \{P\}\ c\ \left\{Q, [\vec{Q'}]\right\}}{\left\{P_{\text{pure}} \wedge P\right\}\ c\ \left\{Q, [\vec{Q'}]\right\}}$$

$$\textsc{Extract-Exists}\ \frac{\forall\,(x:A).\ \{P\}\ c\ \left\{Q, [\vec{Q'}]\right\}}{\{\exists x:A.\ P\}\ c\ \left\{Q, [\vec{Q'}]\right\}} \qquad \textsc{Seq-Assoc}\ \frac{\{P\}\ c_1;(c_2;c_3)\ \left\{Q, [\vec{Q'}]\right\}}{\{P\}\ (c_1;c_2);c_3\ \left\{Q, [\vec{Q'}]\right\}}$$

Fig. 7. Derived rules from C Hoare logic

2.3 Inversion rules and weakest preconditions

The VST program logic is higher-order, i.e., assertions can quantify over assertions, so that one can state the following inversion lemmas, which have already been proven in VST.[5]

Lemma 1 (Inversion on sequencing). *If* $\{P\}\ c_1;c_2\ \left\{Q, [\vec{Q'}]\right\}$, *then*

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

Lemma 2 (Inversion on if-branching). *If* $\{P\}$ if (b) c_1 else c_2 $\left\{Q, [\vec{Q'}]\right\}$, *then*

$$P \vDash \exists P' : \textit{assert},\ P' \wedge \left(\{P' \wedge b \neq 0\}\ c_1\ \left\{Q, [\vec{Q'}]\right\}\right) \wedge \left(\{P' \wedge b = 0\}\ c_2\ \left\{Q, [\vec{Q'}]\right\}\right)$$

It is worth mentioning that the normal postcondition appearing in the inversion on sequencing is VST's representation of weakest precondition, i.e.

$$\text{wp}\left(c, Q, \left[\vec{Q'}\right]\right) \triangleq \exists R : \text{assert}.\ R \wedge \left(\{R\}\ c\ \left\{Q, [\vec{Q'}]\right\}\right).$$

This definition of weakest precondition satisfies basic properties like the following:

Theorem 3. $P \vDash \textit{wp}\left(c, Q, \left[\vec{Q'}\right]\right)$ *if and only if* $\{P\}\ c\ \left\{Q, [\vec{Q'}]\right\}$.

Thus, lemma 1 can be restated as:

$$\{P\}\ c_1;c_2\ \left\{Q, [\vec{Q'}]\right\}\ \text{iff.}\ \{P\}\ c_1\ \left\{\text{wp}\left(c_2, Q, \left[\vec{Q'}\right]\right), [\vec{Q'}]\right\}.$$

We utilize higher-order assertions in our VST-A development.

[5] Readers who knew that VST's separation Hoare logic was proved sound with a semantic proof in a shallow-embedded style may be surprised that it is possible to prove inversion lemmas. But in fact, several years ago the VST-Floyd developers layered a deep-embedded Hoare logic over the shallow-embedded Hoare logic just so that useful lemmas of this kind can be supported.

2.4 Forward symbolic execution

VST's forward verification tactics enable users to obtain the strongest postcondition of a sequence of primary statements automatically. For example, in the verification of path (3) in Fig. 3, the symbolic assignment executor $\mathrm{ae}(P, c)$ for precondition P and statement c can compute the following (where ll is the linked-list predicate):

$$\begin{aligned} P := \quad & l = \mathrm{rev}(l_1)\ x\ l_2\ \wedge\ \mathtt{v} \mapsto (x, u)\ *\ \mathrm{ll}\,(\mathtt{w}, l_1)\ *\ \mathrm{ll}\,(u, l_2) \\ c := \quad & \mathtt{t = v\text{->}tail} \\ \mathrm{ae}(P, c) = \quad & l = \mathrm{rev}(l_1)\ x\ l_2 \wedge\ \mathtt{t} = u\ \wedge\ \mathtt{v} \mapsto (x, u)\ *\ \mathrm{ll}\,(\mathtt{w}, l_1)\ *\ \mathrm{ll}\,(u, l_2) \end{aligned}$$

The assignment executor may fail, if the precondition P cannot guarantee that c will run safely or P is not in a good form[6] such that the symbolic executor can execute c, but we have found if users write assertions in their annotations, corresponding to correct proofs, the symbolic executor can run through the entire straightline Hoare triple. Users are left to prove the entailment from the inferred strongest postcondition (of the straight-line code) to the specified postcondition (arising from the "next" annotation, according to the split function). VST-Floyd also provides useful tactics for solving the entailment problem. Residual proof goals may be generated if some entailments cannot be automatically proven, and users can write their own flexible Coq proof scripts to address them. In summary, with the help of VST-Floyd, the back-end verification of the split results obtained by VST-A can be largely automated.

3 VST-A FRONT END

3.1 Annotated C programs and internal representations

VST-A requires users to describe C function specifications and C programs' functional correctness proofs by writing annotations in C programs. Specifically, a function specification in VST-A is always in a /*@ With ... Require ... Ensure ... */ form and

$$\text{With } [\vec{x} : \vec{A}] \text{ Require } P(\vec{x}) \text{ Ensure } Q(\vec{x})$$

represents a parameterized pre-/postcondition, i.e. it states that for any list of values $\vec{x}$ of type $\vec{A}$, if the initial program state satisfies $P(\vec{x})$ then the C function can be safely executed (no C undefined behavior will happen). If it terminates, the ending state satisfies $Q(\vec{x})$.

For functional correctness proofs, users can describe the main proof skeleton by inserting assertions (including loop invariants) and "given" annotations in the source program. We formally define this annotated C language (Fig. 8), namely ClightA, and implement a front-end parser that converts annotated C programs into the ClightA abstract syntax. Compared with the Clight syntax, ClightA has two new components: assertions and ExGiven structures.

As for assertions, users can insert them anywhere by writing /*@ Assert ... */ in the source program, which is directly converted into a leaf node in the ClightA syntax tree. Annotating loop structures with invariants is *not compulsory* in VST-A,[7] but users can still write loop invariants as /*@ `Inv` ... */ in C source files, to distinguish from assertions before loops. Our front-end automatically converts such annotations into the general syntax of ClightA.

As for the ExGiven structures, users can use a combination of /*@ Assert ∃ `x`, ... */ and /*@ `Given x` */ to represent logical variable introduction in a Hoare logic proof. In a typical

[6]Generally speaking, to ensure that an assignment executor $\mathrm{ae}(P, c)$ always succeeds, P should be in the form of a symbolic heap assertion. When c is a load/store statement, there should be an explicit mapsto predicate for the manipulated variables in the separating conjunction clauses. Otherwise, one might need to apply the rule of consequence (and prove an entailment) to put P into that form.

[7]That is, instead of an invariant at the beginning of the loop body, it may be more convenient to write assertion(s) elsewhere in the loop body, sufficient to break the control flow into straight-line segments; see §3.4.

$$\begin{array}{rrl}
\text{assertion :} & P := & \cdots \\
\text{ClightA statements :} & C := & c_p \mid C_1; C_2 \mid \text{if } (e)\ C_1 \text{ else } C_2 \mid \text{loop } (C_2)\ C_1 \\
 & \mid & \text{skip} \mid \text{break} \mid \text{continue} \mid \text{return} \\
 & \mid & \text{assert } P \mid \text{ExGiven } x : A,\ \{P(x)\}\ C \\
\text{Annotation erasing :} & C\Downarrow\ \in & \text{Clight statement}
\end{array}$$

Fig. 8. ClightA: abstract C language for annotated programs

goal-directed proof strategy, one can extract existential variables from the precondition into the proof context (see Extract-Exist in Fig. 7). Then assertions that appear later in the focused proof can refer to the extracted variables as ordinary Coq assumptions. VST-A supports this proof method by defining the "ExGiven $x : A,\ \{P(x)\}\ C$" syntax. The syntax indicates that assertion P is existentially quantified by logical variable x. Moreover, the inner ClightA statement is also quantified by x, so within the (annotated) assertions of C, one can mention x.

Fig. 9 is a comparison between annotated C programs and ClightA syntax, showing how our front-end parser translates annotations into AST constructions.

```
/*@ Inv P1 */                     loop (skip) {
while (b) {                         assert(P1);
  c1;                               if (b) { skip; } else { break; }
  /*@ Assert ∃ x, P2(x)             c1;
      Given x */                    ExGiven x, P2(x) {
  c2;                                 c2;
  /*@ Assert P3(x) */                 assert(P3(x));
  c3;                                 c3;
}                                 }}
```

Fig. 9. Annoated C program v.s. ClightA syntax

3.2 Expressiveness: describing Hoare logic proofs in VST-A

It is not surprising that the ClightA language is expressive enough to describe the main structures of Hoare logic proofs. Hoare logic proof rules decompose the verification target into smaller ones (e.g., as in Fig. 6 in §2). Describing such proofs in VST-A is natural. For example, to apply the sequence rule Semax-Seq, we can insert the middle condition as an assertion annotation into the C program.

3.3 Expressiveness: describing interactive proofs in VST-A

ClightA is also expressive enough to describe interactive proofs used by existing verification tools like VST. In most cases, these tools try to find proof rules to apply using a *goal-directed strategy*. For example:

- In order to verify a Hoare triple of form $\{P\}\mathtt{x} = e; c \left\{Q, \left[\vec{Q'}\right]\right\}$ in VST, its forward symbolic execution tactic applies the sequence rule Semax-Seq and uses the strongest postcondition of P and x = e as the middle condition. VST-A's users do not need to write any annotation to describe such proof strategy.

- In order to verify a Hoare triple of form

$$\{P\}\texttt{while}\ (e)\ \{\ c_1\ \};c_2\left\{Q,\left[\vec{Q}'\right]\right\}$$

  in VST (suppose no break statements appear in c_1), it asks users to provide a loop invariant I and then $I \wedge e = 0$ will be used as the middle condition between the loop and c_2. In VST-A, the proof effort is similar. Users only need to provide such a loop invariant I in annotated C programs.
- In order to verify a Hoare triple whose precondition is existentially quantified, one will typically extract that existential variable into the proof context. The “given” annotation in VST-A describes this proof strategy.

Besides these goal-directed proof steps, users of interactive program-verification tools may use the rule of consequence anywhere in the middle of a proof, to replace a precondition with a weaker assertion (Fig. 10 shows an example). Correspondingly, users of VST-A can write an assertion annotation to “invoke” the rule of consequence.

$$\begin{array}{lcl} \{\mathrm{ll}\,(\mathrm{p},l)\ \wedge\ \mathrm{p} \neq \mathrm{NULL}\} & & \{\exists\, q\, x\, l'.\mathrm{p} \mapsto (x,q) * \mathrm{ll}\,(q,l')\} \\ \texttt{p -> head = 0} & \text{can be rewritten as} & \texttt{p -> head = 0} \\ \{\exists l'.\mathrm{ll}\,(\mathrm{p},l')\} & & \{\exists l'.\mathrm{ll}\,(\mathrm{p},l')\} \end{array}$$

Fig. 10. Example of replacing the precondition with an existentially quantified assertion. The predicate $\mathrm{ll}\,(\mathrm{p},l)$ means that a **l**inked **l**ist starts at address p representing (in the heads of its cons cells) the sequence l. The two preconditions above are provably equivalent.

In summary, typical interactive proofs are structural, following the syntax tree of the C statements (with local uses of existential variable extraction or the rule of consequence). The ClightA language is able to describe such structural proofs.

3.4 Expressiveness: describing nonstructural proofs in VST-A

ClightA can also describe nonstructural proofs, those that do not follow the C syntax tree. Allowing users to write nonstructural proofs is a considerable convenience. For example, in order to verify a Hoare triple of form

$$\{P\}\texttt{if}\ (e)\ \{\ c_1\ \}\ \texttt{else}\ \{c_2\};c_3\left\{Q,\left[\vec{Q}'\right]\right\}$$

in an interactive verification tool, users will probably be asked to provide a join condition after the if-statetment (a precondition for c_3). In some cases this is appropriate and convenient, but sometimes it is both difficult and unnecessary. Here are some typical scenarios.

- The if-then branch c_1 is the break statement. In this case, it suffices to prove

$$\{P \wedge e \neq 0\}c_1\ \{\bot, [Q_{\mathrm{brk}}, \bot, \bot]\}\ \text{ and }\ \{P \wedge e = 0\}c_2;c_3\left\{Q,\left[\vec{Q}'\right]\right\},$$

  and users may hope to symbolically execute c_2 so that the middle condition between c_2 and c_3 can be generated instead of manually provided.[8]
- The statement c_3 is very short. In this case, verifying c_3 twice can be less work than writing down a join condition.

[8]VST users do not need to provide a join condition in such cases since 2018. VST implemented this feature by adding a built-in program transformation to its verification tactics.

- Proving c_3 functionally correct needs very different proof strategy when e has a different boolean value. In this case, a join condition does not help to reduce the workload: in effect, one is case-splitting on e and then proving c_3 twice.

Similarly, in order to verify a Hoare triple of form

$$\{P\}\texttt{while}\ (e)\ \{\ c_1\ \}; c_2 \left\{Q, \left[\vec{Q}'\right]\right\}$$

where break statements appear in some branches in the loop body c_1, a join condition is needed (in addition to a loop invariant) in an interactive proof, since an execution may leave the loop by a break statement or by falsifying the loop condition e. In VST-A, users can choose to provide a join condition (the proof will be structural) or not to provide a join condition (the proof will be nonstructural).

Certain kinds of generalized loop invariants lead to nonstructural proofs. Consider a red-black tree (RBT) algorithm that reestablishes red-black invariants after insertion. Fig. 11 is a straightforward textbook implementation of this algorithm [Cormen et al. 2022]. When the rotation on line 15 is done, the loop exits immediately, since the assignment statement on line 14 ensures that the loop condition will be evaluated to false in the next iteration. If we were to write a loop invariant on line 3 before the loop condition is checked, we then need to both state the RBT bottom-up fixing invariant and describe the case in which the RBT has been fixed by the final rotation and should exit immediately. One may instead expect a single invariant on line 5, and reason separately about the control flow where the loop exits after a rotation. Such proof strategy is unavailable in common goal-directed verifiers, in which loop invariants are compulsory, but it is supported by VST-A.

```
 1  void insert_balance(struct tree *p, struct tree *root) {
 2    ... // pre−processing code omitted
 3    /*@ Inv: ... (RBT invariant) \/ ... (loop exit property) */
 4    while (p != root && p−>parent−>color != RED) {
 5      /*@ Assert: ... (RBT invariant) */
 6      struct tree *p_par = p−>parent, *p_gpar = p_par−>parent;
 7      if (p_par == p_gpar−>left) {   // p's parent is a left child
 8        struct tree *p_uncle = p_gpar−>right;
 9        if (p_uncle−>color == RED) {
10          p_par−>color = BLACK; p_uncle−>color = BLACK;
11          p_gpar−>color = RED; p = p_gpar;
12        } else {
13          if (p == p_par−>right) { p = p_par; left_rotate(p, root); }
14          p−>parent−>color = BLACK;  p_gpar−>color = RED;
15          right_rotate(p−>parent−>parent, root);
16      } } else { ... }                  // dual case where p's parent is a right child, omitted
17    }}
```

Fig. 11. The fix-up function of red-black tree insertion

4 CONTROL FLOW SPLITTING AND SOUNDNESS

One of the most important components in VST-A is the verified split function which reduces an entire C program's functional correctness to a series of straightline Hoare triples, based on the annotations. Intuitively, this split function is a CFG-based computation (like our demonstration in Fig. 4), and its soundness must ultimately relate to C's small-step semantics—any execution trace of the program statement can be decomposed into these separated paths. However, we choose to prove this soundness theorem directly using VST's program logic *(Verifiable C)*, instead of proving it indirectly by first showing the execution trace decomposition lemma that we mentioned above. Our main consideration is: it is nontrivial to formally establish a theoretical connection between a program logic and an operational semantics, especially when a complicated program logic for a realistic programming language with a lot of subtleties are considered. VST has already done that once—Verifiable C's soundness proof takes 60K lines of Coq definitions and proofs. If we would choose to prove the split function sound using small-step semantics as intermediate proof steps, we might need to develop similar lengthy proofs.

Even in applications to other languages and other operational semantics, it will be useful to build annotation verifiers on top of program logics in the way that we present here, rather than try to prove soundness directly from operational semantics. Proved-sound verification tools tend to be based on similar program logics (at least in terms of the core rules targeted by our split function: the sequence rule, the consequence rule, etc.). But they may be based on quite different styles of operational semantics (e.g. imperative HOL [Bulwahn et al. 2008] uses big step semantics and CompCert Clight uses small step semantics), or they may (like VST [Appel et al. 2014] or Iris [Jung et al. 2018]) incorporate modal impredicativity.

Since most Hoare logic proof rules are syntax-oriented, we implement our split function through recursion on ClightA syntax tree (§4.2) and then we prove it sound by induction (§4.3). In the rest part of this section, we will start from defining the Coq type of split results (§4.1).

4.1 The type of split result

As illustrated in Fig. 12, control flow paths between the assertions can be divided into four classes:

(1) head paths — control flow paths from the precondition to an internal assertion;
(2) tail paths — control flow paths from an internal assertion to the postcondition;
(3) full paths — control flow paths between two internal assertions;
(4) assertion-free paths — control flow paths from the precondition to the postcondition (we call them *assertion-free* since they pass through no assertions inside the annotated program).

Formally, the split result is a record that consists of "head/tail paths", "full paths" and "assertion-free paths", which are essentially a list of *basic program statements* $\vec{c_b}$ annotated with one single assertion, two assertions, and no assertions, respectively. A basic statement can either be a primary Clight statement, c_p, or a special statement, assume e, that represents an if-condition (positively or negatively) in the control flow.

Recall that in the VST program logic, a Hoare triple has multiple postconditions for different kinds of program exits (i.e., exit by break, by continue, by return,or normal fall-through). Correspondingly, the split result also makes distinctions among the different exits. Thus, in the definition of our intermediate split result, the record contains one set of "full paths" between the annotated assertions, one set of "head paths" from the entry point to the internal assertions, four sets of "tail paths" from the internal assertions to the four different kinds of exits, and four sets of "assertion-free paths" from the entry point to the four different kinds of exits. To handle existential variables in their scope, full paths can be universally quantified. With these fields, the split result record can sufficiently reveal all control flow information in a ClightA program. Fig. 13 shows the definition.

By supplementing "head/tail paths" or "assertion-free paths" with the pre-/postconditions, we can interpret the split result into a collection of closed Hoare triples as hypotheses of split's soundness theorem (these hypotheses are illustrated in Fig. 12 and formally defined in Fig. 14):

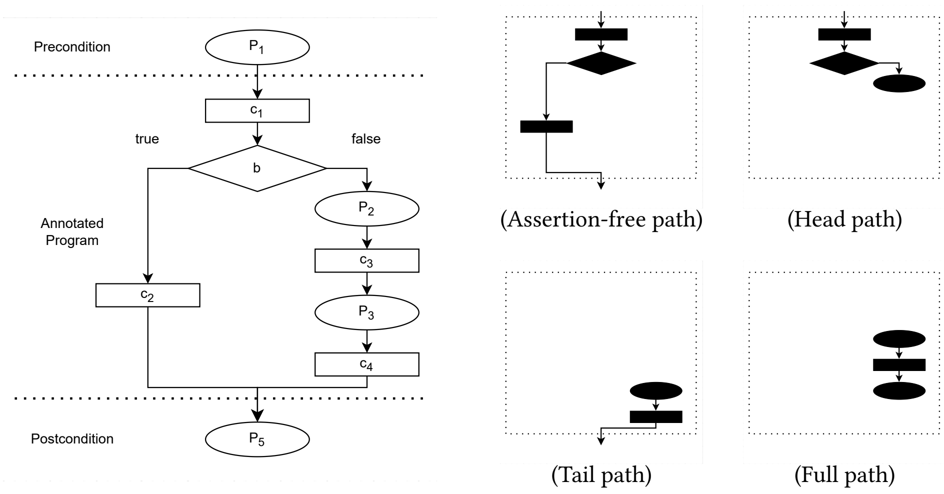

Fig. 12. Control flow graphs v.s. paths in split results

$$\begin{array}{rrcl}
\text{Basic statement :} & c_b & := & c_p \mid \text{assume } e \\
\text{Assertion-free paths :} & p_{-} & := & \vec{c_b} \\
\text{Head paths :} & p_{\dashv} & := & \vec{c_b}\dashv\{P\} \\
\text{Tail paths :} & p_{\vdash} & := & \{P\}\vdash\vec{c_b} \\
\text{Full paths :} & p_{\vdash\dashv} & := & \{P_1\}\vdash\vec{c_b}\dashv\{P_2\} \mid \forall(x:A).\, p_{\vdash\dashv}
\end{array}$$

$$\text{split}(C) = \left\{ \begin{array}{l} \boldsymbol{p}_{-}^{\text{nor}}, \boldsymbol{p}_{-}^{\text{brk}}, \boldsymbol{p}_{-}^{\text{con}}, \boldsymbol{p}_{-}^{\text{ret}}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv} \end{array} \right\}, \text{ where } \begin{array}{l} \boldsymbol{p}_{-}^{\text{nor}}, \boldsymbol{p}_{-}^{\text{brk}}, \boldsymbol{p}_{-}^{\text{con}}, \boldsymbol{p}_{-}^{\text{ret}} \subseteq \text{Assertion-free paths}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}} \subseteq \text{Tail paths}, \\ \boldsymbol{p}_{\dashv} \subseteq \text{Head paths},\ \boldsymbol{p}_{\vdash\dashv} \subseteq \text{Full paths} \end{array}$$

Fig. 13. The type of split results

THEOREM 4 (SOUNDNESS). *For any ClightA program C and pre-/post-conditions P, Q, Q_{brk}, Q_{con} and Q_{ret}, if* $\text{split}(C) = \{\boldsymbol{p}_{-}^{nor}, \boldsymbol{p}_{-}^{brk}, \boldsymbol{p}_{-}^{con}, \boldsymbol{p}_{-}^{ret}, \boldsymbol{p}_{\vdash}^{nor}, \boldsymbol{p}_{\vdash}^{brk}, \boldsymbol{p}_{\vdash}^{con}, \boldsymbol{p}_{\vdash}^{ret}, \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv}\}$ *and*

(a) *all straightline Hoare triples from the precondition P to internal assertions are provable,*
(b) *all straightline Hoare triples from internal assertions to the postconditions $\vec{Q}$ are provable,*
(c) *all straightline Hoare triples between internal assertions are provable,*
(d) *all straightline Hoare triples from the precondition P to the postconditions $\vec{Q}$ are provable,*

i.e. (defined in Fig. 14),

(a) $\text{HeadHypo}(P, \boldsymbol{p}_{\dashv})$,
(b) $\text{TailHypo}(Q, \boldsymbol{p}_{\vdash}^{nor})$, $\text{TailHypo}(Q_{\text{brk}}, \boldsymbol{p}_{\vdash}^{brk})$, $\text{TailHypo}(Q_{\text{con}}, \boldsymbol{p}_{\vdash}^{con})$ *and* $\text{TailHypo}(Q_{\text{ret}}, \boldsymbol{p}_{\vdash}^{ret})$,
(c) $\text{FullHypo}(\boldsymbol{p}_{\vdash\dashv})$,
(d) $\text{AssnFreeHypo}(P, Q, \boldsymbol{p}_{-}^{nor})$, $\text{AssnFreeHypo}(P, Q_{\text{brk}}, \boldsymbol{p}_{-}^{brk})$, $\text{AssnFreeHypo}(P, Q_{\text{con}}, \boldsymbol{p}_{-}^{con})$ *and* $\text{AssnFreeHypo}(P, Q_{\text{ret}}, \boldsymbol{p}_{-}^{ret})$,

then $\{P\}\ C\Downarrow\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}$, *where $C\Downarrow$ represents the result of erasing all annotations from C.*

Noticing that all of the control flows in a function body should end with a return statement, we directly use the following corollary in VST-A.

$$
\begin{aligned}
&\text{HeadHypo}(P, \vec{c_b}\dashv\{Q\}) = \{P\}\ \vec{c_b}\ \left\{Q, [\vec{\top}]\right\}\\
&\text{HeadHypo}(P, \boldsymbol{p}_{\dashv}) = \textstyle\bigwedge_{p\in\boldsymbol{p}_{\dashv}}\ \text{HeadHypo}(P, p)\\[1em]
&\text{AssnFreeHypo}(P, Q, \vec{c_b}) = \{P\}\ \vec{c_b}\ \left\{Q, [\vec{\top}]\right\}\\
&\text{AssnFreeHypo}(P, Q, \boldsymbol{p}_{-}) = \textstyle\bigwedge_{p\in\boldsymbol{p}_{-}}\ \text{AssnFreeHypo}(P, Q, p)\\[1em]
&\text{TailHypo}(Q, \{P\}\vdash\vec{c_b}) = \{P\}\ \vec{c_b}\ \left\{Q, [\vec{\top}]\right\}\\
&\text{TailHypo}(Q, \boldsymbol{p}_{\vdash}) = \textstyle\bigwedge_{p\in\boldsymbol{p}_{\vdash}}\ \text{TailHypo}(Q, p)\\[1em]
&\text{FullHypo}(\{P\}\vdash\vec{c_b}\dashv\{Q\}) = \{P\}\ \vec{c_b}\ \left\{Q, [\vec{\top}]\right\}\\
&\text{FullHypo}(\forall x.\ p_{\vdash\dashv}) = \forall x.\ \text{FullHypo}(p_{\vdash\dashv})\\
&\text{FullHypo}(\boldsymbol{p}_{\vdash\dashv}) = \textstyle\bigwedge_{p\in\boldsymbol{p}_{\vdash\dashv}}\ \text{FullHypo}(p)
\end{aligned}
$$

Fig. 14. Hypotheses of the soundness theorem

Corollary 5. *For any ClightA program C and pre-/post-conditions P and Q, if*

$$\text{split}(C) = \left\{\emptyset, \emptyset, \emptyset, \boldsymbol{p}_{-}^{ret}, \emptyset, \emptyset, \emptyset, \boldsymbol{p}_{\vdash}^{ret}, \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv}\right\}$$

and (a) $\text{HeadHypo}(P, \boldsymbol{p}_{\dashv})$, *(b)* $\text{TailHypo}(Q, \boldsymbol{p}_{\vdash}^{ret})$, *(c)* $\text{FullHypo}(\boldsymbol{p}_{\vdash\dashv})$, *and (d)* $\text{AssnFreeHypo}(P, Q, \boldsymbol{p}_{-}^{ret})$, *then* $\{P\}C\Downarrow\{Q\}$.

Remark. CompCert Clight does not have an assume statement. We choose to encode the assume statement into Clight AST, and encode straightline Hoare triples into VST Hoare triples, so that VST-A's users can directly use VST's tactics to prove those straightline triples. Specifically,

$$\textbf{assume}\ e\ \triangleq \textbf{if}\ (e)\ \text{skip}\ \textbf{else}\ \text{break};$$

$$\{P\}\ c_1; c_2; \ldots; c_n\ \{Q\} \triangleq \{P\}\ c_1; c_2; \ldots; c_n\ \{Q, [\top, \bot, \bot]\}\,.$$

We proved:

Lemma 6. *For straightline Hoare triples,* $\{P\}$ ***assume*** $e;\ c\ \{Q\}$ *if and only if* $\{P \wedge e \neq 0\}\ c\ \{Q\}$.

4.2 Split function

The core split function is defined by recursion on the abstract syntax tree of the input ClightA program. Note that the split function is a partial function, since we will not try to compute the reduction result if there is an assertion-free loop in the CFG.

Base cases (Fig. 15). For primary statements c_p, the normal assertion-free path set (the $\boldsymbol{p}_{-}^{\text{nor}}$ field) is a singleton of the statement itself, i.e. $\{[c_p]\}$, and all other path sets are empty. For a break statement, the assertion-free break-exit path set is a singleton of an empty list of basic statements, i.e. $\{[\,]\}$. The split results of continue and return are similar. For the assertion annotation assert P, the split result has only one head path $\{[\,]\dashv\{P\}\}$ and one normal tail path $\{\{P\}\vdash[\,]\}$.

Recursion cases. Using sequential composition as an example, all "full paths" between the assertions in $C_1; C_2$ can be divided into three classes (Fig. 16): (1) paths that completely fall in the CFG of C_1; (2) paths that completely fall in the CFG of C_2; and (3) paths that combine two parts, one of which is a "tail path" of C_1 and the other of which is a "head path" of C_2. Such "tail paths" and "head paths" of an assertion-annotated C program can also be recursively computed. We show our complete definition in Fig. 17.

$$\mathrm{split}(c_p) = \left\{\begin{array}{l} \{[c_p]\}, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array}\right\} \mathrm{split}(\mathrm{break}) = \left\{\begin{array}{l} \emptyset, \{[\,]\}, \emptyset, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array}\right\} \mathrm{split}(\mathrm{return}) = \left\{\begin{array}{l} \emptyset, \emptyset, \emptyset, \{[\,]\} \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array}\right\}$$

$$\mathrm{split}(\mathrm{continue}) = \left\{\begin{array}{l} \emptyset, \emptyset, \{[\,]\}, \emptyset \\ \emptyset, \emptyset, \emptyset, \emptyset \\ \emptyset, \emptyset \end{array}\right\} \quad \mathrm{split}(\mathrm{assert}\ P) = \left\{\begin{array}{l} \emptyset, \emptyset, \emptyset, \emptyset \\ \{\{P\}{-}[\,]\}, \emptyset, \emptyset, \emptyset \\ \{[\,]{-}\{P\}\}, \emptyset \end{array}\right\}$$

Fig. 15. Split function for basic statements

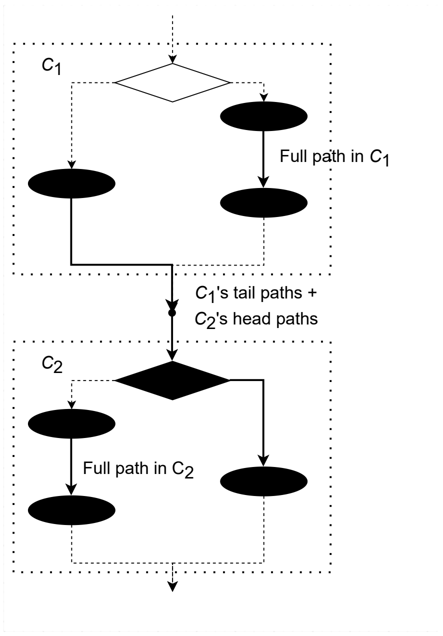

Fig. 16. Different full paths in $C_1; C_2$

If

$$\mathrm{split}(C_1) = \left\{\begin{array}{l} \boldsymbol{p}_{-}^{\mathrm{nor}}, \boldsymbol{p}_{-}^{\mathrm{brk}}, \boldsymbol{p}_{-}^{\mathrm{con}}, \boldsymbol{p}_{-}^{\mathrm{ret}}, \\ \boldsymbol{p}_{\vdash}^{\mathrm{nor}}, \boldsymbol{p}_{\vdash}^{\mathrm{brk}}, \boldsymbol{p}_{\vdash}^{\mathrm{con}}, \boldsymbol{p}_{\vdash}^{\mathrm{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv} \end{array}\right\}$$

$$\mathrm{split}(C_2) = \left\{\begin{array}{l} \boldsymbol{q}_{-}^{\mathrm{nor}}, \boldsymbol{q}_{-}^{\mathrm{brk}}, \boldsymbol{q}_{-}^{\mathrm{con}}, \boldsymbol{q}_{-}^{\mathrm{ret}}, \\ \boldsymbol{q}_{\vdash}^{\mathrm{nor}}, \boldsymbol{q}_{\vdash}^{\mathrm{brk}}, \boldsymbol{q}_{\vdash}^{\mathrm{con}}, \boldsymbol{q}_{\vdash}^{\mathrm{ret}}, \\ \boldsymbol{q}_{\dashv}, \boldsymbol{q}_{\vdash\dashv} \end{array}\right\},$$

then

$$\mathrm{split}(C_1; C_2) = \left\{\begin{array}{l} \boldsymbol{p}_{-}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{nor}}, \\ \boldsymbol{p}_{-}^{\mathrm{brk}} \cup \boldsymbol{p}_{-}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{brk}}, \\ \boldsymbol{p}_{-}^{\mathrm{con}} \cup \boldsymbol{p}_{-}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{con}}, \\ \boldsymbol{p}_{-}^{\mathrm{ret}} \cup \boldsymbol{p}_{-}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{ret}}, \\ \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cup \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{nor}}, \\ \boldsymbol{p}_{\vdash}^{\mathrm{brk}} \cup \boldsymbol{q}_{\vdash}^{\mathrm{brk}} \cup \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{brk}}, \\ \boldsymbol{p}_{\vdash}^{\mathrm{con}} \cup \boldsymbol{q}_{\vdash}^{\mathrm{con}} \cup \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{con}}, \\ \boldsymbol{p}_{\vdash}^{\mathrm{ret}} \cup \boldsymbol{q}_{\vdash}^{\mathrm{ret}} \cup \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{ret}}, \\ \boldsymbol{p}_{\dashv} \cup \boldsymbol{p}_{-}^{\mathrm{nor}} \cdot \boldsymbol{q}_{\dashv}, \\ \boldsymbol{p}_{\vdash\dashv} \cup \boldsymbol{q}_{\vdash\dashv} \cup \boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{\dashv} \end{array}\right\}.$$

Fig. 17. The definition of $\mathrm{split}(C_1; C_2)$

In this definition, we use $\cdot$ to represent the concatenation of two paths, and overload this notation to connect two sets of paths:

$$\begin{array}{rcllll} p_{\vdash} \cdot q_{\dashv} & = & \{P\}{-}(\vec{c_b} +\!\!+\, \vec{c_b}\,'){-}\{Q\} & \text{where } p_{\vdash} = \{P\}{-}\vec{c_b} & \text{and } q_{\dashv} = \vec{c_b}\,'{-}\{Q\} \\ p_{-} \cdot q_{\dashv} & = & (\vec{c_b} +\!\!+\, \vec{c_b}\,'){-}\{Q\} & \text{where } p_{-} = \vec{c_b} & \text{and } q_{\vdash} = \vec{c_b}\,'{-}\{Q\} \\ p_{\vdash} \cdot q_{-} & = & \{P\}{-}(\vec{c_b} +\!\!+\, \vec{c_b}\,') & \text{where } p_{\vdash} = \{P\}{-}\vec{c_b} & \text{and } q_{-} = \vec{c_b}\,' \\ p_{-} \cdot q_{-} & = & \vec{c_b} +\!\!+\, \vec{c_b}\,' & \text{where } p_{-} = \vec{c_b} & \text{and } q_{-} = \vec{c_b}\,' \\ \boldsymbol{p} \cdot \boldsymbol{q} & = & \{p \cdot q \mid p \in \boldsymbol{p},\ q \in \boldsymbol{q}\} & & \end{array}$$

Computing split(if (b) C_1 else C_2) simply adds assume statements to the head of all head paths and assertion-free paths in the two if-branches, and returns the union of the two split results. Computing split(loop (C_2) C_1) is similar. Detailed definitions can be found in the extended version of the paper and our Coq development.

Handling logical variables (Fig. 18). The focus of computing split(ExGiven $x : A$, $\{P(x)\}$ C_1) is to handle the logical variable x. (1) The ExGiven structure has an existentially quantified precondition $P(x)$ in the head. Therefore, there are no assertion-free paths in the split result, and the result of the

$$
\begin{array}{l}
\text{If } \mathsf{split}(C_1) = \left\{\begin{array}{l} \boldsymbol{p}_{-}^{\text{nor}}, \boldsymbol{p}_{-}^{\text{brk}}, \boldsymbol{p}_{-}^{\text{con}}, \boldsymbol{p}_{-}^{\text{ret}}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\!\dashv} \end{array}\right\}, \\
\text{then } \mathsf{split}(\text{ExGiven } x : A,\ \{P(x)\}\ C_1) = \\
\quad \left\{\begin{array}{l}
\emptyset, \emptyset, \emptyset, \emptyset \\
\left\{\{\exists x : A.\ Q\}\vdash\vec{c_b}\ |\{Q\}\vdash\vec{c_b} \in \boldsymbol{p}_{\vdash}^{\text{nor}}\right\} \cup \{\{\exists x : A.\ P(x)\}\vdash[]\} \cdot \boldsymbol{p}_{-}^{\text{nor}} \\
\left\{\{\exists x : A.\ Q\}\vdash\vec{c_b}\ |\{Q\}\vdash\vec{c_b} \in \boldsymbol{p}_{\vdash}^{\text{brk}}\right\} \cup \{\{\exists x : A.\ P(x)\}\vdash[]\} \cdot \boldsymbol{p}_{-}^{\text{brk}} \\
\left\{\{\exists x : A.\ Q\}\vdash\vec{c_b}\ |\{Q\}\vdash\vec{c_b} \in \boldsymbol{p}_{\vdash}^{\text{con}}\right\} \cup \{\{\exists x : A.\ P(x)\}\vdash[]\} \cdot \boldsymbol{p}_{-}^{\text{con}} \\
\left\{\{\exists x : A.\ Q\}\vdash\vec{c_b}\ |\{Q\}\vdash\vec{c_b} \in \boldsymbol{p}_{\vdash}^{\text{ret}}\right\} \cup \{\{\exists x : A.\ P(x)\}\vdash[]\} \cdot \boldsymbol{p}_{-}^{\text{ret}} \\
\{[]\dashv\{\exists x : A.\ P(x)\}\}, \\
\left\{\forall x : A.\ \{P\}\vdash\vec{c_b}\dashv\{Q\}|\ \vec{c_b}\dashv\{Q\} \in \boldsymbol{p}_{\dashv}\right\} \cup \left\{\forall x : A.\ \{Q\}\vdash\vec{c_b}\dashv\{R\}|\{Q\}\vdash\vec{c_b}\dashv\{R\} \in \boldsymbol{p}_{\vdash\!\dashv}\right\}
\end{array}\right\}
\end{array}
$$

Fig. 18. The definition of $\mathsf{split}(\text{ExGiven } x : A,\ \{P(x)\}\ C_1)$

head paths is a singleton of $[]\dashv\{\exists x : A.\ P(x)\}$. (2) The tail paths in ExGiven $x : A,\ \{P(x)\}\ C_1$ can either be paths from the precondition $\exists x : A.\ P(x)$ to exits of C_1 or tail paths of C_1 itself. When these tail paths connect to head paths later, the postconditions of those head paths will not be in the scope of x. Thus, we existentially quantify over the variable x in all those tail paths' preconditions now, i.e. if $\{Q(x)\}\vdash\vec{c_b}$ is a tail path of C_1, then $\{\exists x : A.\ Q(x)\}\vdash\vec{c_b}$ is a tail path of ExGiven $x : A,\ \{P(x)\}\ C_1$. (3) In full paths, we need to unify the existential variable x in P with those in the head paths that are split from C_1, so that x can be shared among the pre-/post-conditions of each combined full path. Full paths in C_1 ($\boldsymbol{p}_{\vdash\!\dashv}$) are also collected after adding a universal binder x to the result. In our Coq development, we implement this definition using Coq dependent types. We put technique details in the extended version of the paper.

4.3 Proof of soundness

We prove theorem 4 by induction over ClightA syntax trees. For the convenience of presentation, we use $\text{AllHypo}(P, \vec{Q}, \mathsf{split}(C))$ to represent all ten hypotheses of this soudnenss theorem: In order words, the soundness theorem says: $\text{AllHypo}(P, \vec{Q}, \mathsf{split}(C))$ implies $\{P\}\ C \Downarrow \left\{\vec{Q}\right\}$.

$$
\begin{array}{l}
\text{AllHypo}\left(P, Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}], \left\{\begin{array}{l} \boldsymbol{p}_{-}^{\text{nor}}, \boldsymbol{p}_{-}^{\text{brk}}, \boldsymbol{p}_{-}^{\text{con}}, \boldsymbol{p}_{-}^{\text{ret}}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\!\dashv} \end{array}\right\}\right) \\
\triangleq \quad \begin{array}{l}
\text{AssnFreeHypo}(P, Q, \boldsymbol{p}_{-}^{\text{nor}}) \quad \wedge \quad \text{AssnFreeHypo}(P, Q_{\text{brk}}, \boldsymbol{p}_{-}^{\text{brk}}) \quad \wedge \\
\text{AssnFreeHypo}(P, Q_{\text{con}}, \boldsymbol{p}_{-}^{\text{con}}) \quad \wedge \quad \text{AssnFreeHypo}(P, Q_{\text{ret}}, \boldsymbol{p}_{-}^{\text{ret}}) \quad \wedge \\
\text{TailHypo}(Q, \boldsymbol{p}_{\vdash}^{\text{nor}}) \quad \wedge \quad \text{TailHypo}(Q_{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{brk}}) \quad \wedge \\
\text{TailHypo}(Q_{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{con}}) \quad \wedge \quad \text{TailHypo}(Q_{\text{ret}}, \boldsymbol{p}_{\vdash}^{\text{ret}}) \quad \wedge \\
\text{HeadHypo}(P, \boldsymbol{p}_{\dashv}) \ \wedge\ \text{FullHypo}(\boldsymbol{p}_{\vdash\!\dashv}).
\end{array}
\end{array}
$$

In this proof, only induction steps about sequential compositions, if-statements and loops are interesting. We describe the main idea of proving $\mathsf{split}(C_1; C_2)$ sound in this section. The proofs for if-statements and loops are similar and can be found in the extended version of the paper.

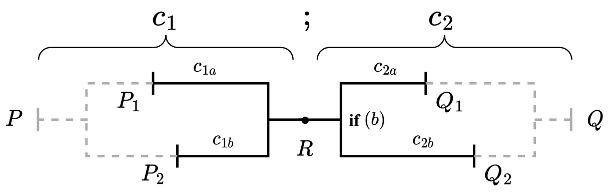

Fig. 19. Soundness proof example of sequential composition

Fig. 19 shows an example of sequential composition with precondition P, postcondition Q, and annotations P_1, P_2, Q_1, Q_2. The split function will generate these straightline Hoare triples:

$$\begin{array}{lll} \{P_1\} & c_{1a}; \mathsf{assume}\ b; c_{2a} & \{Q_1\} \\ \{P_1\} & c_{1a}; \mathsf{assume}\ !b; c_{2b} & \{Q_2\} \\ \{P_2\} & c_{1b}; \mathsf{assume}\ b; c_{2a} & \{Q_1\} \\ \{P_2\} & c_{1b}; \mathsf{assume}\ !b; c_{2b} & \{Q_2\} \\ & \cdots & \end{array}$$

In order to prove $\{P\}\ C_1 \Downarrow; C_2 \Downarrow\ \{Q\}$, we need to find an intermediate assertion R such that $\{P\}\ C_1 \Downarrow\ \{R\}$ and $\{R\}\ C_2 \Downarrow\ \{Q\}$ (according to Semax-Seq). These two judgments can be established by induction hypothesis and the following four Hoare triples:

$$\{P_1\}\ c_{1a}\,\{R\} \quad \{R\}\ \mathsf{assume}\ b; c_{2a}\,\{Q_1\} \quad \{P_2\}\ c_{1b}\,\{R\} \quad \{R\}\ \mathsf{assume}\ !b; c_{2b}\,\{Q_2\}$$

These requirements can be simply satisfied if we let R be

$$\mathrm{wp}(\mathsf{assume}\ b; c_{2a},\ Q_1) \wedge \mathrm{wp}(\mathsf{assume}\ !b; c_{2b},\ Q_2).$$

In the general case, we instantiate R to the conjunction of the weakest preconditions of all head paths and assertion-free paths in $\mathsf{split}(C_2)$. Using VST's higher order logic, this middle condition can be written as:

$$R \triangleq \left\{ \begin{array}{l} \exists R.\ R \wedge \mathrm{HeadHypo}(R, \boldsymbol{q}_{\dashv}) \\ \quad \wedge \mathrm{AssnFreeHypo}(R, Q, \boldsymbol{q}_{-}^{\mathrm{nor}}) \\ \quad \wedge \mathrm{AssnFreeHypo}(R, Q_{\mathrm{brk}}, \boldsymbol{q}_{-}^{\mathrm{brk}}) \\ \quad \wedge \mathrm{AssnFreeHypo}(R, Q_{\mathrm{con}}, \boldsymbol{q}_{-}^{\mathrm{con}}) \\ \quad \wedge \mathrm{AssnFreeHypo}(R, Q_{\mathrm{ret}}, \boldsymbol{q}_{-}^{\mathrm{ret}}) \end{array} \right\}$$

According to the induction hypothesis, it suffices to prove the following two propositions.

$$\mathrm{AllHypo}(P, R, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}], \mathsf{split}(C_1)) \tag{1}$$

$$\mathrm{AllHypo}(R, Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}], \mathsf{split}(C_2)) \tag{2}$$

The proof of (2) is simple, which can be justified by the following lemma in VST.

Lemma 7. *For any program c and postcondition $\vec{Q}$, $\left\{\exists P : \mathit{assert}.\ P \wedge \left(\{P\}\ c\ \left\{\vec{Q}\right\}\right)\right\}\ c\ \left\{\vec{Q}\right\}$ holds.*

For proposition 1, the inversion lemma for sequencing (Lemma 1) has already shown that the weakest precondition of the second statement can serve as the intermediate assertion for the sequential composition. Based on Lemma 1, we can prove a corresponding inversion lemma on the $\cdot$ operator for each type of path in the split results.

Proposition 8 (Inversion lemmas for split results).

(1) *If* $\mathrm{AssnFreeHypo}(P, Q, p_{-} \cdot q_{-})$*, then* $\mathrm{AssnFreeHypo}(P, \exists R.\ R \wedge \mathrm{AssnFreeHypo}(R, Q, q_{-}), p_{-})$
(2) *If* $\mathrm{HeadHypo}(P, p_{-} \cdot q_{\dashv})$*, then* $\mathrm{AssnFreeHypo}(P, \exists R.\ R \wedge \mathrm{HeadHypo}(R, q_{\dashv}), p_{-})$

(3) *If* $\mathrm{TailHypo}(Q, p_{\vdash} \cdot q_{-})$*, then* $\mathrm{TailHypo}(Q, \exists R.\ R \wedge \mathrm{AssnFreeHypo}(R, Q, q_{-}), p_{\vdash})$
(4) *If* $\mathrm{FullHypo}(p_{\vdash} \cdot q_{\dashv})$*, then* $\mathrm{TailHypo}(\exists R.\ R \wedge \mathrm{HeadHypo}(R, q_{\dashv}), p_{\vdash})$

According to the conjunction rule, AssnFreeHypo and TailHypo from Proposition 8 still hold if we combine all of those weakest preconditions of C_2's partial paths (i.e. those preconditions are such that $\mathrm{AssnFreeHypo}(R, Q, q_{-})$ and $\mathrm{HeadHypo}(R, q_{\dashv})$). Formally, Proposition 8 can be extended into the following form by the conjunction rule.

Proposition 9 (Grouped Inversion lemmas).
(1) *If* $\mathrm{AssnFreeHypo}(P, Q, \boldsymbol{p}_{-} \cdot \boldsymbol{q}_{-})$*, then* $\mathrm{AssnFreeHypo}(P, \exists R.\ R \wedge \mathrm{AssnFreeHypo}(R, Q, \boldsymbol{q}_{-}), \boldsymbol{p}_{-})$
(2) *If* $\mathrm{HeadHypo}(P, \boldsymbol{p}_{-} \cdot \boldsymbol{q}_{\dashv})$*, then* $\mathrm{AssnFreeHypo}(P, \exists R.\ R \wedge \mathrm{HeadHypo}(R, \boldsymbol{q}_{\dashv}), \boldsymbol{p}_{-})$
(3) *If* $\mathrm{TailHypo}(Q, \boldsymbol{p}_{\vdash} \cdot \boldsymbol{q}_{-})$*, then* $\mathrm{TailHypo}(Q, \exists R.\ R \wedge \mathrm{AssnFreeHypo}(R, Q, \boldsymbol{q}_{-}), \boldsymbol{p}_{\vdash})$
(4) *If* $\mathrm{FullHypo}(\boldsymbol{p}_{\vdash} \cdot \boldsymbol{q}_{\dashv})$*, then* $\mathrm{TailHypo}(\exists R.\ R \wedge \mathrm{HeadHypo}(R, \boldsymbol{q}_{\dashv}), \boldsymbol{p}_{\vdash})$

Theorem 10 (Conjunction Rule). *If Hoare triples* $\{P\}\ c\ \left\{Q_1, [\vec{Q}_1']\right\}$ *and* $\{P\}\ c\ \left\{Q_2, [\vec{Q}_2']\right\}$ *are derivable, then* $\{P\}\ c\ \left\{Q_1 \wedge Q_2, [\vec{Q}_1' \wedge \vec{Q}_2']\right\}$ *is derivable.*

Next, we also use the conjunction rule to combine the weakest preconditions of the different kinds of paths and prove proposition 1, which completes the soundness proof of $\mathsf{split}(C_1; C_2)$.

However, the conjunction rule is not ubiquitous among the Hoare logic variants proposed in the literature: for example, the current VST program logic cannot derive the conjunction rule. We will leave the discussion of the conjunction rule to §5. For now, we assume that the conjunction rule holds, so that Proposition 9 and the proof of $\mathsf{split}(C_1; C_2)$ soundness can hold.

5 CONJUNCTION RULE AND PRECISENESS

The conjunction rule is natural in traditional Hoare logics and separation logics for sequential programs, but some extensions to the logics will make the conjunction rule inadmissible. In this section, we more extensively discuss why the conjunction rule is required by our soundness proof (§5.1, §5.2). To make the conjunction rule admissible in VST-A (§5.3, §5.4, §5.5), we identify a new notion of preciseness to restrict the function specifications being called during verification. We also discuss the trade-offs of using conjunction rules and precise function specifications, and we suggest some future directions for improvement (§5.6).

5.1 A small example

Suppose we would like to prove the Hoare triple

$$\{P\}\ c_1;\ \mathbf{if}\ (b)\ c_2\ \mathbf{else}\ c_3\ \{R\} \tag{3}$$

given that the following split results hold (here, we assume that c_1, c_2 and c_3 are primary statements):

$$\{P\}\ c_1;\ \mathbf{assume}\ b;\ c_2\ \{R\} \tag{4}$$

$$\{P\}\ c_1;\ \mathbf{assume}\ !b;\ c_3\ \{R\} \tag{5}$$

By inversion on sequential composition (lemma 1), proposition (3), (4) and (5) are equivalent to:

$$\{P\}\ c_1\ \{\mathrm{wp}(\mathbf{if}\ (b)\ c_2\ \mathbf{else}\ c_3,\ R)\}$$
$$\{P\}\ c_1\ \{\mathrm{wp}(\mathbf{assume}\ b;\ c_2,\ R)\}$$
$$\{P\}\ c_1\ \{\mathrm{wp}(\mathbf{assume}\ !b;\ c_3,\ R)\}$$

Furthermore, by inversion (Lemmas 1 and 2) and properties of **assume** (Lemma 6), $\mathrm{wp}(\mathbf{if}\ (b)\ c_2\ \mathbf{else}\ c_3,\ R)$ is equivalent to

$$\mathrm{wp}(\mathbf{assume}\ b;\ c_2,\ R)\ \wedge \mathrm{wp}(\mathbf{assume}\ !b;\ c_3,\ R)$$

Thus, the split function's soundness on the example above can be reduced to an instance of the conjunction rule.

5.2 Unsoundness when the conjunction rule is inadmissible

So far, we have seen a tight connection between the conjunction rule and the split function's soundness — our soundness proof uses the conjunction rule (§4.3) and a very simple instance of this soundness theorem can be reduced to an instance of the conjunction rule (§5.1). But what will happen to split's soundness if the conjunction rule is not admissible? Consider Hoare logic with ghost updates [Krebbers et al. 2017a] as an example. Ghost states are "logical states" that help with the program's proof, and they particularly useful for verifying concurrent programs.[9] When users prove programs with ghost states, they can apply a ghost update when they use the consequence rule, see Semax-Conseq-Ghost below. Here, $\Rrightarrow P$ says: there is at least one possible ghost update which makes the state satisfy P.

$$\textsc{Semax-Conseq-Ghost}\ \frac{P_1 \vDash\ \Rrightarrow P_2 \qquad R_2 \vDash\ \Rrightarrow R_1 \qquad \vec{R_2'} \vDash\ \Rrightarrow \vec{R_1'} \qquad \{P_2\}\ c\ \left\{R_2, [\vec{R_2'}]\right\}}{\{P_1\}\ c\ \left\{R_1, [\vec{R_1'}]\right\}}$$

The conjunction rule is not admissible in this logic — if the proofs of $\{P\}\ c\ \{Q\}$ and $\{P\}\ c\ \{Q'\}$ use different and conflicting ghost updates, $\{P\}\ c\ \{Q \wedge Q'\}$ cannot be valid since two conflicting ghost updates cannot happen simultaneously.

In this logic, our split function is unsound. This loss of soundness is not determined by the way we prove soundness in §4.3 but by the framework we propose to first split the program into individual paths, which are then verified separately. Consider the following annoated program:

```
1   /*@ Assert g ↦ A */
2   f();
3   x = nondetermined_0_or_1();
4   if (x) {
5       /*@ Assert g ↦ B_1 ∧ x = 1 */
6       f1();
7   } else {
8       /*@ Assert g ↦ B_0 ∧ x = 0 */
9       f0();
10  }
11  /*@ Assert g ↦ C_1 ∧ x = 1 ∨ g ↦ C_0 ∧ x = 0 */
```

in which g is a ghost location for storing the status of the following STS (state transition system) [Turon et al. 2013]: $A \longrightarrow A_0,\ A \longrightarrow A_1,\ A_0 \longrightarrow B_0,\ A_1 \longrightarrow B_1,\ B_0 \longrightarrow C_0, B_1 \longrightarrow C_1$. We assume that f(),

[9]Ghost states are not the same as the "ghost variables" of traditional Hoare logics. Ghost variables are logical variables that were introduced to relate old values of variables to current values, and to relate current values to abstract mathematical values. VST and Iris support ghost variables using ordinary Coq variables; those ghost variables are fully compatible with our VST-A program decomposition, to support data abstraction and modular verification [Beringer 2021]. Examples of such variables in Fig. 1 are l, l_1, u, x, l_2.

f0(), and f1()'s specifications are:

$$\begin{aligned}
&\text{for any } b \in \textbf{bool},\\
&\ \{g \mapsto (\text{if } b \text{ then } A_1 \text{ else } A_0)\}\ f()\ \{g \mapsto (\text{if } b \text{ then } B_1 \text{ else } B_0)\}\\
&\{g \mapsto B_0\}\ f0()\ \{g \mapsto C_0\}\\
&\{g \mapsto B_1\}\ f1()\ \{g \mapsto C_1\}\,.
\end{aligned}$$

In this example, all straightline Hoare triples in split's result are provable, especially the following two triples about f():

$\{g \mapsto A\}$ f(); x = nondetermined_0_or_1(); **assume** x; $\{g \mapsto B_1 \wedge \text{x} = 1\}$
$\{g \mapsto A\}$ f(); x = nondetermined_0_or_1(); **assume** !x; $\{g \mapsto B_0 \wedge \text{x} = 0\}$

For the first triple, we can choose to take the $A \longrightarrow A_1$ step in ths STS before calling f(). For the second triple, we can choose to take the $A \longrightarrow A_0$ step in ths STS before calling f(). However, the whole Hoare triple is not provable since we cannot determine the value of x before x = nondetermined_0_or_1() is executed. That breaks split's soundness.

5.3 VST-A's design choice and proof strategy

In the current design of VST-A, we focus on sequential program verification and disallow all ghost updates. VST-A uses a more restricted variant of the VST program logic. This variant is still proved sound w.r.t. CompCert Clight semantics and the most significant change is that the ghost update operator is removed from the consequence rule.

Despite the removal of ghost updates, users are still able to write unrestricted higher-order predicates and prove many complex sequential programs in VST-A. We derive the conjunction rule (theorem 10) from this stronger logic by induction over Clight abstract syntax tree. Our inductive proof steps are all Hoare-logic-based, and we believe that such a proof strategy is (1) easier to formalize in Coq, and (2) in fact more general than a semantic-based proof, since it is independent of how the soundness of the Hoare logic was proved w.r.t. its semantic model.

Consider the induction step for $c = c_1; c_2$. By applying Lemma 1 to the premises we obtain the following:

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

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

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

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

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

Using Extract-Exists and Extract-Prop, it suffices to prove:

$$\{R\}\ c_2\ \left\{Q_1, [\vec{Q}_1']\right\}, \{R\}\ c_2\ \left\{Q_2, [\vec{Q}_2']\right\} \Rightarrow \{R\}\, c_2 \left\{Q_1 \wedge Q_2, \left[\vec{Q}_1' \wedge \vec{Q}_2'\right]\right\}$$

which immediate follows the induction hypothesis of c_2.

For other induction steps (e.g. for if-statements and for loops), the proof idea is somewhat similar in that their corresponding inversion lemmas are first applied, then the induction hypotheses can be used to complete the proof. For control flow statements (break, continue and return), the proof is trivial. Thus, we are only left to derive the conjunction rules for primary statements.

5.4 The conjunction rule for memory stores

Consider a simple store statement and corresponding proof rule in VST.

$$\forall\, v.\ \Big\{\ p \mapsto v * (p \mapsto v' \mathrel{-\!\!*} R)\ \Big\}\ *p = v'\ \{R\} \tag{6}$$

The precondition states that the heap can be split into two parts. One part is a singleton heap, in which the nonaddressable variable p is a pointer to the old value v, and it is described by the mapsto predicate $x \mapsto v$. The other part should satisfy the postcondition R when joined with a singleton heap that stores the new value v'.

Note that in this specification, the old value v is universally quantified. If we want to conjoin the postconditions of two Hoare triples about $*p = v'$, we need to unify the two different instantiations of v into one. To be specific, the following property needs to hold:

Proposition 11 (Preciseness of store).

$$\begin{array}{rl} & (\exists\, v_1.\, p \mapsto v_1 * (p \mapsto v' \mathrel{-\!\!*} R_1)) \land \\ & (\exists\, v_2.\, p \mapsto v_2 * (p \mapsto v' \mathrel{-\!\!*} R_2)) \\ \vDash & \exists\, v.\, p \mapsto v * (p \mapsto v' \mathrel{-\!\!*} R_1 \land R_2) \end{array}$$

In VST-A, we prove this through semantic-model-level reasoning based on its underlying memory model. Then, the conjunction rule for such simplified store statements can be logically derived. We leave the detailed proof for the extended version of the paper.

In VST, assignment statements $e_1 := e_2$ include (1) assigning a value to a nonaddressable variable, (2) loading a value from memory to a nonaddressable variable and (3) storing a value in memory. VST's higher-order assertion language also supports separation logic predicates with fractional permissions. In VST-A, we have proved that the conjunction rule holds for all primary set/load/store operations.

5.5 The conjunction rule for function calls

The store rule mentioned above can be treated as a specification for a function call that contains only a single store instruction. We have shown that Proposition 11 is an important property for deriving the conjunction rule. In this section, we generalize this property, which we refer to as *precise function specifications*, to derive the conjunction rule for function calls.

As previously mentioned, VST-A function specification has a form of

$$\text{With } [\vec{a} : \vec{A}].\ \text{Require } \{\lambda\vec{y}.P\}\ \text{Ensure } \{\lambda r.Q\}\,,$$

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 $\vec{A}$ represents the type of logical values to be shared between P and Q. Therefore, both P and Q will also be abstracted over a set of logical variables $\vec{a}$ typed $\vec{A}$. We define the precise function specifications as follows:

Definition 12 (Precise function specification). With $[\vec{a} : \vec{A}]$. Require $\{\lambda\vec{y}.P\}$ Ensure $\{\lambda r.Q\}$ is precise, if for any formal parameters $\vec{b}$, return value r and assertions R_1, R_2, it holds that

$$\begin{array}{rl} & \Big(\exists\vec{x}_1 : \vec{A}.\ P\,\vec{b}\,\vec{x}_1 * (Q\,r\,\vec{x}_1 \mathrel{-\!\!*} R_1)\Big) \land \Big(\exists\vec{x}_2 : \vec{A}.\ P\,\vec{b}\,\vec{x}_2 * (Q\,r\,\vec{x}_2 \mathrel{-\!\!*} R_2)\Big) \\ \vDash & \exists\vec{x} : \vec{A}.\ P\,\vec{b}\,\vec{x} * (Q\,r\,\vec{x} \mathrel{-\!\!*} R_1 \land R_2) \end{array}$$

In VST-A's program logic (a variant of VST's program logic), the function call rule, Semax-Call, requires the callee function specification to be precise.

$$\textsc{Semax-Call}\ \frac{\begin{array}{c}\phi = \text{With } [\vec{x} : \vec{A}].\ \text{Require } \{\lambda \vec{y}.P\}\ \text{Ensure } \{\lambda r.Q\} \\ f \text{ satisfies } \phi \qquad \phi \text{ is a precise specification}\end{array}}{\left\{\vec{e} \Downarrow \vec{b} \wedge \exists \vec{x} : \vec{A}.\ P\,\vec{b}\,\vec{x} * (Q\,\mathsf{a}\,\vec{x} \mathrel{-\!\!*} R)\right\}\ \mathsf{a} := f(\vec{e})\ \left\{R, [\vec{\bot}]\right\}}$$

The notion of "precise function specification" that we propose is defined with respect to an operation (e.g. a store statement or a function call), while traditionally, "preciseness" is defined for a predicate. A typical example of a precise predicate is the $p \mapsto v$ predicate used in Semax-Store.

Definition 13 (Precise predicate). A predicate P is precise if for any Q_1, Q_2,

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

In fact, our definition of precise function specifications includes a common use case of precise predicates. As proved by Gotsman et al. [2011] and Vafeiadis [2011], if the conjunction rule is expected in concurrent separation logic with locks, the resource invariant for locks should be a precise predicate. Consider the following specification for a release operation for lock l:

$$\forall\, l\ \pi.\ \{\pi \text{ is readable} \wedge \mathsf{locked}_\pi(l, R) * R\}\ \mathsf{release}(l)\ \{\mathsf{unlocked}_\pi(l, R)\}$$

where R is the predicate that describes the locked memory. We use locked_π and $\mathsf{unlocked}_\pi$ to denote whether this memory is owned by the current thread or not; π is a fractional permission. It is clear that if the resource invariant R is a precise predicate, then one can show that this specification is a precise specification. The precise function specification also makes a difference in the sequential setting we are considering. One example of a non-precise function specification is:

```
void foo (int * p)
  /*@ With $q$    Require p ↦ 0 * $F(q)$    Ensure p ↦ 1 * $F(q)$ */
  { * p = 1; return; }
```

Here F is a predicate dependent on a logical variable q describing the frames that the function foo does not modify. It is possible to find such F that holds for two different instantiations of q, and breaks the precise function specification requirement.

In our Coq development, we find that, for most function specifications, we can derive that for any formal parameters $\vec{b}$, logical variables $\vec{x_1}, \vec{x_2}$:

$$\left(P\,\vec{b}\,\vec{x_1} * \mathsf{True}\right) \wedge \left(P\,\vec{b}\,\vec{x_2} * \mathsf{True}\right) \vDash \vec{x_1} = \vec{x_2} \tag{7}$$

Then, to prove a specification precise, it is sufficient to show that P is a precise predicate. A typical example is the store predicate "$\mapsto$". Clearly:

$$\forall\, p\ v_1\ v_2.\ ((\mathsf{p} \mapsto v_1 * \mathsf{True}) \wedge (\mathsf{p} \mapsto v_2 * \mathsf{True})) \vDash v_1 = v_2 \tag{8}$$

For linked lists' representation predicate ll, we also have:

$$\forall\, p\ l_1\ l_2.\ ((\mathrm{ll}\,(\mathsf{p}, l_1) * \mathsf{True}) \wedge (\mathrm{ll}\,(\mathsf{p}, l_2) * \mathsf{True})) \vDash l_1 = l_2 \tag{9}$$

Moreoever, property (7) is composable. For example, in order to prove:

$$((\mathsf{p} \mapsto x_1 * \mathrm{ll}(x_1, l_1)) * \mathsf{True}) \wedge ((\mathsf{p} \mapsto x_2 * \mathrm{ll}(x_2, l_2)) * \mathsf{True}) \vDash x_1 = x_2 \wedge l_1 = l_2$$

we can first use (8) to derive $x_1 = x_2$, then use after (9) to derive $l_1 = l_2$.

It is well-known that precise predicates are also composable, thus we developed several extensible automatic tactics in VST-A to help users prove the precise specification by combining property (7) and predicates' preciseness.[10]

We believe that most C functions have specifications that are naturally precise. Non-precise specifications are usually caused by separation logic conjuncts that describe inaccessible memory slices to the function, such as the $F(q)$ proposition in the example above. In practical verification tasks, one can usually remove these conjuncts using the "frame rule" and obtain a precise specification. The pruned part of the specification can be expressed elsewhere in the program, where the memory slice is really used.

5.6 Discussion and future work

In our current design of VST-A, we require the conjunction rule to be derivable in the logic to ensure the splitting algorithm's soundness. We do not consider our current design choice as a fatal problem for future extensions. Here are some potential research directions that may support ghost updates or remove the restriction of only using precise specifications in the future.

Verify the sequential fragments of a concurrent program. As was discussed in §5.2, the existence of the conjunction rule forbids the use of ghost updates. However, there are ways to verify the sequential fragment of a concurrent program with VST-A. We have proved the following property:

Theorem 14. *If* $\{P\}\, c_1; c_2 \models \left\{R, [\vec{R'}]\right\}$*, where* $\models$ *indicates that the triple is derivable from the original VST logic that supports ghost state updates, then there exists* $Q, \vec{Q'}$ *such that* $\{P\}\ c_1\ \left\{Q, [\vec{Q'}]\right\}$ *and* $\{Q\}\, c_2 \models \left\{R, [\vec{R'}]\right\}$.

The triple about c_1 is derivable from the stronger program logic in VST-A. We can apply the current VST-A framework to the sequential fragment c_1, and leave the rest of the program to be verified by the full-power VST interactive verifier.

New annotations for ghost actions. Another possible direction is to improve the capability of annotations in terms of expressing proofs. For example, in future versions of VST-A, we could allow users to write either explicit ghost commands in their programs, or write two consecutive assertions where the latter can be derived from the former with a ghost update. As a result, we do not need to worry about the possibility of having two conflicting ghost updates simultaneously.

Additional annotations for function calls. As was discussed in §5.5, in order to derive the conjunction rule for function calls, the callee should meet the requirement of precise function specification. However, this requirement could be removed if the logical variables of the specification are explicitly instantiated. This avoids the possibility that proofs of multiple straightline Hoare triples containing the same function call could actually instantiate those logical variables differently, causing issues in proving the conjunction rule. Intuitively, this is like putting two assertions before and after the function call statement as pseudo "join points" in the control flow, ensuring that every function call would only appear in one unique path in the split result. Then, split's soundness would be trivial for function calls.

Prophecy variables. In our counterexample in §5.2, the verification target is unprovable but will become provable if we are allowed to insert a prophecy variable [Abadi and Lamport 1988; Jung

[10]In several cases only weaker equivalence relations can be derived from the conjunction. Our tactics also support proof automation for that, if this relation implies the equality of the specification

et al. 2020] into the program. In general, we may be able to prove split's soundness with the help of prophecy variables even if ghost updates are permitted in the logic.

6 EVALUATION

In this section we evaluate VST-A in various aspects, including verification effort, verification time, and statistics of our development. We also draw a comparison between VST-A and VST.

Verification effort using VST-A. We test VST-A using several sets of programs and present the statistics about verification effort in Table 1. VST-A proofs are divided into two parts: the annotated programs (which describe the main idea of the proof in a way easier to read than VST proofs) and Coq verification of straightline Hoare triples (similar to corresponding parts of VST proofs). Therefore, we count the lines of annotations and proofs separately, in the Specification and Assertion columns and the Proof column, respectively.

Our benchmarks include some measured by Sammler et al. [2021], including linked lists and binary search trees. We also include a slightly larger example—a small-step interpreter of a toy imperative language, which includes a number of if-branches. It can be seen from our evaluation that without providing additional assertions in the program code, VST-A is still able to split and verify the program.

				VST-A		VST
Program	Functions	Code	Spec	Assert	Proof	Proof
Basics	8	78	56	5	84	156
Singly linked list	18	350	85	55	969	1212
Doubly linked list	4	95	16	16	171	213
Binary search tree	4	115	44	23	202	364
Interpreter	5	337	47	0	423	653

Table 1. Number of C functions, lines of C code (without annotations), VST-A specification lines annotated in the C comments, assertion annotations in the C comments, Coq proofs in VST-A, versus Coq specifications and proofs in VST.

We also conduct a comparison in the proof effort between VST-A and VST. The last column in Table 1 shows the lines of proofs for verifying the same function in VST; compare this to the "Specification+Assertion+VST-A Proof" columns. The proof lines do not count auxiliary predicate definitions and lemmas since they are the same in VST and VST-A.

Not manifest in the line-count, but still important, is that the VST user must learn to use several different tactics for different forms of control flow, and each of these has several options depending on which assertions are supplied (e.g., if-postcondition, for-loop continue assertion, for-loop break assertion, etc.). In contrast, VST-A's control-flow splitting takes care of all of this, leaving the user to learn only the straight-line *forward* and *forward_call* tactics. In some cases, splitting makes the Coq proof easier to automate. For example, verifying the interpreter example in Table 1 using VST-A only needs to use *forward* to handle assumes repeatedly, but this proof strategy cannot be used in the corresponding VST proof.

To summarize, we believe that to verify the same program, the manual effort of VST-A is no more than that of VST, while in the meantime VST-A can provide more intuitive and readable proofs with annotations in the C source program.

Verification time for different phases in VST-A. The verification time is shown in Table 2. Since the splitting process in VST-A is a computationally proven sound function, it is not surprising that the

Program	Reduction	Common	Avg. compile	Avg. verify	Max. verify	VST verify
Basics	0.239	12.0	1.7×14	4.8×14	8.1	29.1
SLL	0.087	11.0	1.7×57	6.9×57	12.2	159.2
DLL	0.064	16.0	1.7×13	7.1×13	13.5	48.5
BST	0.069	11.0	1.7×18	7.4×18	12.2	54.8
Interpreter	0.090	25.0	1.7×45	20.8×45	41.9	222.9

Table 2. Verification time for different phases (in seconds). Reduction: time used for parsing C and generating straightline paths in Coq. Common: time used for compiling common definitions and theorems. Avg. compile: the average time used for compiling the straightline Hoare triple definitions of all generated paths × the number of paths. Avg. verify: the average time used for verifying each path × the number of paths. Max. verify: the maximum time used for verifying each path. VST verify: the time used for verifying the same programs in VST.

reduction time is short. For different verification tasks, users need to provide different separation logic predicates and lemmas to specify and prove the program. This consitutes a majority of the compilation time, which is also needed by a VST proof.

A significant difference between VST-A and VST is that in VST-A each straightline Hoare triple's proof can be developed as a separate lemma, residing in distinct files, facilitating parallel checking and compilation. Therefore, we present the average time for these phases and the number of paths. In VST, the correctness theorem of a function needs to be proved as a whole, which is more difficult to parallelize.

In addition to parallelized compilation, the separation of straightline Hoare triple proofs also makes it easier to maintain the correctness of the proofs when the program is modified. Only those paths that are affected by the modification need to be re-verified. We conducted several experiments and the results are shown in Table 3.

Program	Function	Paths	Changes	Changed paths
Singly linked list	append	5	modify the pre-condition	1
Singly linked list	rev_append	4	change the loop invariant	3
Singly linked list	reverse	$4 \longrightarrow 3$	remove an assertion in the loop	$2 \longrightarrow 1$
Binary search tree	lookup	5	modify the post-condition	2
Interpreter	eval	21	change the code in one branch	1

Table 3. Case study of the number of paths that need to be re-verified when the program is modified. $4 \longrightarrow 3$ indicates that the program modification changed the number of paths.

Statistics of the development of VST-A. We present the line of codes statistics for our development in VST-A below. Our development in VST-A includes the following parts:

- Coq formalization of the restricted fragment of VST program logic, including the logical rules, auxiliary lemmas, and the conjunction rule proof: 6,236 lines
- Coq formalization of the VST-A framework, including the split algorithm, and its soundness proof: 6,287 lines
- Modification to the CompCert C parser to parse annotated C programs: 1.87% of change
- Modification to the VST-Floyd lemmas and automation tactics to support forward symbolic execution in the restricted fragment of VST program logic: 6.74% of change

- OCaml development that provides an efficient implementation of the split algorithm: 3,176 lines in addition to original CompCert

7 RELATED WORK

7.1 Traditional annotation verifiers

Many annotation verifiers work by reducing annotated programs to SMT assertion entailments: these include Frama-C, Dafny, VeriFast, Viper, Hip/Sleek. Some of those systems use a specialized intermediate language for verification, connectable to several SMT back-ends and to several programming-language front-ends; for example Frama-C uses Why3 [Filliâtre and Paskevich 2013] and Dafny uses Boogie [Barnett et al. 2006]. Modern SMT solvers effectively and efficiently solve many of the resulting entailments. These annotation verifiers are, in practice, "interactive:" one starts by annotating the program with function specifications and some loop invariants, and the verifier inevitably points out several places where the proof fails—with sufficiently good error messages that the user can adjust the assertions, add new assertions and invariants, and try again, and again. This works well in practice, and it is what we wanted to emulate.

The disadvantage of those systems is in the poverty of their assertion languages. Because SMT (or Why3 or Boogie) accommodates only first-order or near-first-order logics, the rich specification languages of VST or of logics built in Iris cannot be used. In VST one often proves high-level properties about the behavior of programs, in application-specific mathematics that would be difficult to fit into SMT. In Frama-C, Dafny and CN (etc.), some authors work around this by writing higher-level proofs in Coq and program-logic proofs in the annotation verifier, and stapling together the two verifications (in different logics without a common foundation) [Boldo et al. 2014]. That approach could be made more foundational by embedding a semantics of Why3 in Coq [Cohen and Johnson-Freyd 2024], but the assertion language would still be near-first-order.

Another disadvantage of those annotation verifiers is that none has a machine-checked proof of soundness (e.g., w.r.t. an operational semantics). This lack is not inherent, as VST-A demonstrates.

When proving programs in VST or in an Iris-based logic, one generally uses automated solvers to prove entailments, or at least to prove the easy parts and leave residuals for the user. These solvers may be programmed in Coq (using tactics or computational reflection) or may be external (such as SMT). VST-A does not choose a specific solver. Users can choose their own solver to prove the split result correct.

BRiCk [Malecha et al. 2022], used by BedRock Systems Inc. to verify its microkernel/hyperviser, is a program logic for C++, built in Iris, on principles inspired by VST. We expect that our VST-A method would work well in such a C++ program logic.

Tools like F* [Martínez et al. 2019], LiquidHaskell [Vazou et al. 2014, 2018], and ATS [Chen and Xi 2005] have managed to combine higher-order programming with theorem proving in a dependent type system so that rich higher-order properties can be automatically verified in the style of the program's annotations. However, they require users to either write programs in a new domain-specific language or construct the proof as a term in the program. By contrast, VST-A works on the standard (and practical) C programming language while also enabling reasoning with higher-order properties.

We also note the work of sledgehammer [Böhme and Nipkow 2010] and auto2 [Zhan 2016] for proof automation in Isabelle. Sledgehammer relies on SMT solvers, while auto2 builds compositional proof automation using a saturation-based proof automation system in which goal-directed proof strategies can be encoded. Users of auto2 can easily extend auto2 with their own domain-specific proof strategy. Zhan [2018] built an auto2 instance that supports separation logic reasoning for verifying sequential programs. Although auto2 supports flexible saturation-based proof strategies, this specific instance of sequential program verification is mainly goal-directed. VST-A is not goal-directed, and is open to any solver or proof style when verifying entailments, regardless of whether it is, based on an interactive proof a tactic-based solver, or a model checking based one.

7.2 Interactive prover-based program verification

There is no deep reason why annotation verifiers should lack soundness proofs (e.g., Frama-C, Dafny, Verifast) and tactic-based verifiers should have machine-checked soundness proofs (e.g., VST, Iris). We guess that the reason is: such soundness proofs are naturally higher-order, more easily accommodated in the kinds of higher-order logics implemented in proof assistants such as Coq and Isabelle, so it is natural that designers of VST and Iris *also* have their *users* operate in the same proof assistants.

Soundness proofs are important for real-world programming languages, which have many subtle features in their semantics and compilers. Users want what they prove about a program to be consistent with the compiled machine code semantics, so the foundational soundness of VST-A is a real benefit. In this section, we compare VST-A with other foundational tools.

VST and various Iris-based verifiers have invested significantly in increasing proof automation so that users can verify their programs conveniently. However, they all require their users to complete correctness proofs for the entire program in an interactive theorem prover, which is not easy for an ordinary software engineer to learn.

There are also works that build annotation verification into interactive theorem provers and have achieved foundational soundness. RefinedC [Sammler et al. 2021] is an automated and foundational annotation verifier for C programs, that defines a restricted fragment of the Iris logic, Lithium, so that proof searches can be guided by translating the assertion annotations into a Lithium program. DiaFrame [Mulder et al. 2022] is also an automated and foundational tool. It employs a similar structural approach to RefinedC, but is more targeted at proving fine-grained concurrent programs. These tools use tactic-based proof strategy design and achieve some reasonable automation. In other words, a Hoare triple will be reduced to smaller proof goals (and even directly solved) by automatically applying a series of proof tactics, which use proved-sound logic rules or verified single-step symbolic execution. In comparison, VST-A is based on one computational proved-sound reduction function. Thus, developers of VST-A do not need to *decompose* this reduction step into multiple proof tactics, which in the end allows users to describe more flexible proofs using annotated C programs. Here is an example of how reduction is decomposed into tactics. Given an annotated program of form (here, we use a pair of braces to emphasize that sequential composition is right associative in CompCert Clight and in our ClightA syntax):

$$/{*}@\ P\ {*}/\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2;\ \{\ c_3;\ /{*}@\ Q\ {*}/\ c_4\ \}\ /{*}@\ R\ {*}/$$

VST-A will generate 3 straightline Hoare triples:

$$\{P\}\textbf{assume}\ b; c_1; c_3\{Q\} \qquad \{P\}\textbf{assume}\ !b; c_2; c_3\{Q\} \quad \text{and} \quad \{Q\}c_4\{R\}.$$

In order to achieve this in RefinedC's or DiaFrame's tactic-based proof automation system, the system needs to apply the Seq-Assoc rule (see §2.2) first, turning the proof goal into:

$$\{P\}\ \{\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2; c_3\ \}\ ; c_4\ \{R\}$$

and then apply the sequence rule with middle condition Q. After that, one more proof rule[11] is needed for turning **if** (b) c_1 **else** $c_2; c_3$ into **if** (b) $\{\ c_1; c_3\ \}$ **else** $\{\ c_2; c_3\ \}$ so that the Semax-If rule can be used to complete the reduction. However, such tactic-based decomposition is not always easy to design, and it can even be impossible. Especially, it is not obvious how to design tactic-based

[11]In most Iris-based verifiers, this last step is not needed since Iris's symbolic execution can do that implicitly.

proof automation for handling nontraditional loop invariants supported by VST-A. That is, our split function (in effect) performs some nontrivial static analysis and is verified by a nontrivial soundness proof. Besides supporting more flexible proofs, the core split function in VST-A is computation-based so that VST-A can first complete the reduction step very efficiently, and users can then prove straightline Hoare triples manually or using their own domain-specific proof automations. Also, this design of VST-A can better support incremental development.

7.3 Conjunction rule and preciseness

The conjunction rule is naturally sound in traditional Hoare logic [Floyd 1993]. However, for concurrent separation logic with locks, a counterexample that leads to unsoundness [O'Hearn 2004] can be found. As a workaround, De Vilhena et al. [2020] proved a restricted version called the candidate rule, which requires postconditions to be pure (independent of resources, especially ghost resources) to solve their verification problem. However, this rule cannot be applied in our setting. In VST-A, we do not propose alternative rules but prove the conjunction rule on top of a VST logic without ghost updates. As for supporting concurrency, we proposed several possible directions in §5.6.

We are not aware of any similar notions of precise function specifications in the literature as we have defined in this paper. Traditionally, preciseness restrictions are placed on assertion predicates. For example, in concurrent separation logic, the resource invariant should be a precise predicate to make the conjunction rule sound [Gotsman et al. 2011; Vafeiadis 2011]. Our aim is to define a notion of preciseness for specifications, so that the conjunction rule is derivable from the existing logical rules. Compared with traditional preciseness, we showed in §5.5 that our notion of preciseness is more expressive, as it accounts for a pair of pre-/post-conditions for an operation and allows the specification to be quantified by logical variables.

8 CONCLUSION

We have presented VST-A, an annotation verifier that is foundationally verified. VST-A targets a widely used real-world language, C, and supports higher-order assertions in the very rich specification language of VST that includes the full expressive power of Coq. VST-A splits the verification of a large program into verifications of straightline control flow paths separated by assertions. The soundness of this approach requires the conjunction rule to be derivable in the program logic. We have identified a novel notion of precise specifications in the proof of the conjunction rule. Currently, VST-A only supports sequential C program verification, but we have proposed ways to extend VST-A to support concurrency in the future. Our formal annotation language and other major designs are not C-specific, nor are they separation-logic-specific, nor VST-specific. A similar development can be used to design other Hoare-style annotation verifiers for imperative languages.

Comparing to existing foundational program verification tools built in interactive theorem provers, VST-A has the following advantages:

- Annotation-based proof is a more readable way to explain why a program is correct.
- Our annotation-based proof language ClightA is expressive enough to describe nonstructural proofs, which cannot be supported systematically using goal-directed tactic-based proof automation;
- VST-A is easier to use — users only need to write assertions in annotations, and use forward symbolic execution to prove straightline Hoare triples. In comparison, users of other tools like VST and Iris need to use different tactics to handle different program structures like if-conditions, recursions and different loops.

- VST-A reduces proof recompilation time. When a verified program is updated slightly, its correctness proof also needs corresponding updates. In existing interactive verifiers, users must rerun all tactical proof scripts, even though only a small portion needs to be updated. Now, only the part of the program that has been changed and the corresponding proof need to be recompiled, since other parts in split's result are unchanged.

We aim to enhance the verification process of VST-A further. Future work includes the introduction of domain-specific heuristics for automatically manipulating separation logic predicates during symbolic execution and proving separation logic entailments on straightline Hoare triples. We also plan to allow user to specify partial assertions, so that users do not need to write assertions for the entire state of the function throughout the whole program.

ACKNOWLEDGMENTS

This material is based upon work supported (in part) by NSF China 61902240, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0160.

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.* IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

Andrew W. Appel. 2011. Verified software toolchain (Invited talk). In *Lecture Notes in Computer Science*, Vol. 6602 LNCS. Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer, Berlin, 364–387.

Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: The Frama-C Software Analysis Platform. , 56–67 pages. https://doi.org/10.1145/3470569

Lennart Beringer. 2021. Verified Software Units. In *Lecture Notes in Computer Science*, Vol. 12648 LNCS. Springer, Cham, 118–147. https://doi.org/10.1007/978-3-030-72019-3_5

Sascha Böhme and Tobias Nipkow. 2010. Sledgehammer: Judgement day. In *Lecture Notes in Computer Science*, Vol. 6173 LNAI. Springer, Berlin, Heidelberg, 107–121. https://doi.org/10.1007/978-3-642-14203-1_9

Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. 2014. Trusting computations: A mechanized proof from partial differential equations to actual program. *Computers and Mathematics with Applications* 68, 3 (2014), 325–352. https://doi.org/10.1016/j.camwa.2014.06.004

Pierre Boutillier, Stephane Glondu, Benjamin Grégoire, Hugo Herbelin, Pierre Letouzey, Pierre-Marie Pédrot, Yann Régis-Gianas, Matthieu Sozeau, Arnaud Spiwack, and Enrico Tassi. 2014. Coq 8.4 Reference Manual. (jul 2014). https://hal.inria.fr/hal-01114602

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 134–149. https://doi.org/10.1007/978-3-540-71067-7_14

Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of C programs. In *Lecture Notes in Computer Science*, Vol. 6617 LNCS. Springer, Berlin, Heidelberg, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (jun 2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

Chiyan Chen and Hongwei Xi. 2005. Combining programming with theorem proving. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP.* 66–77. https://doi.org/10.1145/1086365.1086375

Wei Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming* 77, 9 (aug 2012), 1006–1036. https://doi.org/10.1016/j.scico.2010.07.004

Joshua M. Cohen and Philip Johnson-Freyd. 2024. A Formalization of Core Why3 in Coq. *Proceedings of the ACM on Programming Languages* 8, POPL (2024).

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. MIT press.

Paulo Emílio De Vilhena, François Pottier, and Jacques Henri Jourdan. 2020. Spy game: Verifying a local generic solver in iris. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371101

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming (ESOP'13)*. Springer, 125–128.

Robert W. Floyd. 1993. *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4 (reprint of a 1967 paper).

Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the conjunction rule in concurrent separation logic. In *Electronic Notes in Theoretical Computer Science*, Vol. 276. 171–190. https://doi.org/10.1016/j.entcs.2011.09.021

Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Naoto Maeda, Takashi Imoto, Rakesh Pothengil, and Mustafa Hussain. 2015. Scalable and scope-bounded software verification in Varvel. *Automated Software Engineering* 22, 4 (dec 2015), 517–559. https://doi.org/10.1007/s10515-014-0164-0

F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. 2005. F-SOFT: Software verification platform. In *Lecture Notes in Computer Science*, Vol. 3576. Springer, Berlin, Heidelberg, 301–306. https://doi.org/10.1007/11513988_31

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Lecture Notes in Computer Science*, Vol. 6617 LNCS. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. *ACM SIGPLAN Notices* 52, 1 (jan 2017), 205–217. https://doi.org/10.1145/3009837.3009855

Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker. In *Lecture Notes in Computer Science*, Vol. 8413 LNCS. Springer, Berlin, Heidelberg, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Lecture Notes in Computer Science*, Vol. 6355 LNAI. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Gregory Malecha, Gordon Stewart, František Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Security & Privacy* 20, 3 (2022), 33–42. https://doi.org/10.1109/MSEC.2022.3158196

Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Lecture Notes in Computer Science*, Vol. 11423 LNCS. Springer Verlag, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2 arXiv:1803.06547

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. (2022), 16. https://doi.org/10.1145/3519939.3523432

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*. Vol. 9583. Springer Verlag, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

Peter W. O'Hearn. 2004. Resources, concurrency, and local reasoning. *Lecture Notes in Computer Science* 2986 (2004), 1–2. https://doi.org/10.1007/978-3-540-24725-8_1

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL, Article 1 (jan 2023), 32 pages. https://doi.org/10.1145/3571194

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings - Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/lics.2002.1029817

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 343–356. https://doi.org/10.1145/2429069.2429111

Viktor Vafeiadis. 2011. Concurrent separation logic and operational semantics. In *Electronic Notes in Theoretical Computer Science*, Vol. 276. Elsevier, 335–351. https://doi.org/10.1016/j.entcs.2011.09.029

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–31. https://doi.org/10.1145/3158141 arXiv:1711.03842

Bohua Zhan. 2016. AUTO2, a saturation-based heuristic prover for higher-order logic. In *International Conference on Interactive Theorem Proving*. Springer, 441–456.

Bohua Zhan. 2018. Efficient Verification of Imperative Programs Using Auto2. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 23–40. https://doi.org/10.1007/978-3-319-89960-2_2

Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2023. VST-A: A Foundationally Sound Annotation Verifier. *CoRR* abs/1909.00097 (2023). arXiv:1909.00097 http://arxiv.org/abs/1909.00097