Assertion annotated program verification with control flow splitting

Litao Zhou
Shanghai Jiao Tong University
Shanghai, China
ltzhou@sjtu.edu.cn

Abstract

We propose a system for decomposing the verification of assertion annotated C programs into simple Hoare triples for sequences of basic statements. In the system, users can write higher order assertions in C programs' comments. Our split function will analyze the control flow and collect all the paths separated by assertions as verification goals automatically. We describe a prototype system implementation VST-A, which is developed based on Verified Software Toolchain (VST)'s deep-embedded separation logic. We implement the splitting function and formally verified its soundness w.r.t. CompCert's Clight semantics in Coq.

CCS Concepts: • **Theory of computation → Separation logic**; • **Program verification**;

Keywords: Program Verification, Separation logic, Annotated Programs, Coq

1 Overview of the framework

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

Figure 1 compares the common verification workflow in VST and our framework VST-A. VST provides a set of Hoare logic rules, which are proved sound with respect to the semantics of the language. To prove the correctness of a program c with respect to a pair of its pre-/post-condition specification P and Q, users need to write a proof script in the interactive theorem prover Coq, by applying the Hoare logic rules, and providing intermediate assertions (e.g. loop invariants) where necessary. The soundness of VST ensures end-to-end correctness of the proof.

In VST-A, we would like to allow users to verify a program by annotating programs with assertions, which constitutes the core wisdom of a program's verification. The verification workflow we advocate is depicted on the right side of Figure 1. Users first write assertions as comments in the source code of the programs, and then a path-splitting algorithm will automatically analyze the control flow between each two assertions and generate a set of paths $\{\{P_i\}c_i\{Q_i\}\}$ for users to verify independently. Since control flow information has been exploited, the commands of each path c_i are simply linear sequences of basic statements, which are left for users to verify. After all paths have been verified, the soundness theorem of VST-A will link them together, and together with the soundness theorem of VST, end-to-end verification is preserved.

The path-splitting function, and the mechanized proof of its soundness theorem are the two major contributions of VST-A against VST. We will introduce the path-splitting algorithm and the proof idea of soundness theorem in Section 2. Section 3 will discuss the "Conjunction rule", which we consider as a fundamental component of this framework.

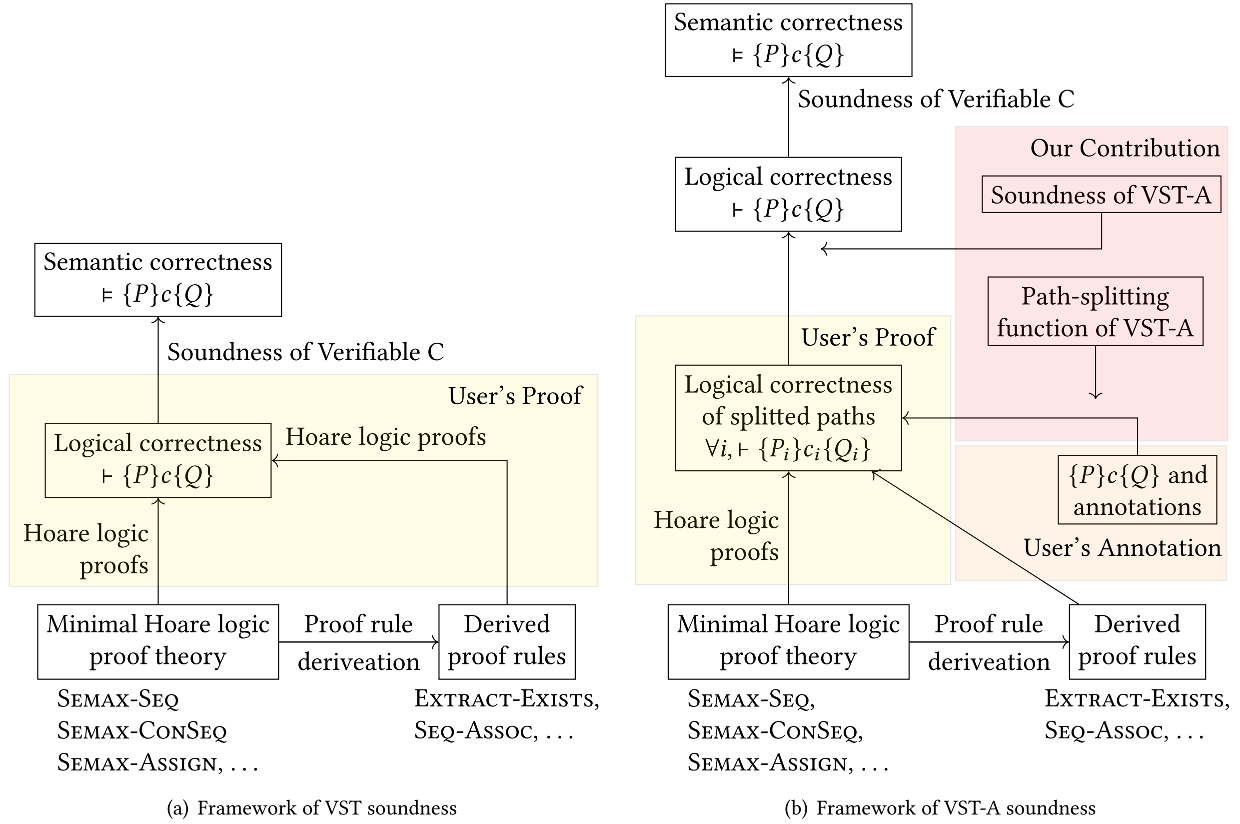
2 Path split and its soundness

2.1 Clight-A: Annotated C language

We define the annotation Clight-A syntax as an extension to the abstract C language of CompCert Clight in Figure 2. The Clight-A syntax provides extra constructors that allow users to insert assertions. The syntax of assertions is the same as defined in VST program logic, which we omit here.

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

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



(a) Framework of VST soundness

(b) Framework of VST-A soundness

Figure 1. Overview of the framework

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

Figure 2. syntax of Clight and Clight-A

$$
\begin{array}{rrcl}
\text{Basic statement :} & c_b & := & c_p \mid e \\
\text{Path atoms :} & p_{-} & := & \vec{c_b} \\
\text{Path atoms with return :} & p_{-}^{\text{ret}} & := & \vec{c_b}; ?e \\
\text{Simple pre-partial path :} & p_{s\dashv} & := & \vec{c_b} \dashv \{\} \\
\text{Complete pre-partial path :} & p_{\dashv} & := & \vec{c_b} \dashv \{P\} \\
\text{Simple post-partial path :} & p_{s\vdash} & := & \{\} \vdash \vec{c_b} \\
\text{Complete post-partial path :} & p_{\vdash} & := & \{P\} \vdash \vec{c_b} \\
 & & | & \forall (x : A).\ p_{\vdash} \\
\text{Simple post-return path :} & p_{s\vdash}^{\text{ret}} & := & \{\} \vdash \vec{c_b}; e? \\
\text{Complete post-return path :} & p_{\vdash}^{\text{ret}} & := & \{P\} \vdash \vec{c_b}; e? \\
 & & | & \forall (x : A).\ p_{\vdash}^{\text{return}} \\
\text{Simple full path :} & p_{s\vdash\dashv} & := & \{\} \vdash \vec{c_b} \dashv \{\} \\
\text{Complete full path :} & p_{\vdash\dashv} & := & \{P_1\} \vdash \vec{c_b} \dashv \{P_2\} \\
 & & | & \forall (x : A).\ p_{\vdash\dashv}
\end{array}
$$

$$
\text{Split result :} \left\{ \begin{array}{l} \boldsymbol{p}_{-}^{\text{nor}}, \boldsymbol{p}_{-}^{\text{brk}}, \boldsymbol{p}_{-}^{\text{con}}, \boldsymbol{p}_{-}^{\text{ret}}, \\ \boldsymbol{p}_{\vdash}^{\text{nor}}, \boldsymbol{p}_{\vdash}^{\text{brk}}, \boldsymbol{p}_{\vdash}^{\text{con}}, \boldsymbol{p}_{\vdash}^{\text{ret}}, \\ \boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv} \end{array} \right\}
$$

Figure 3. syntax of split results

Clight-A syntax C_c is implemented as a dependepent type on a particular simple Clight-A syntax C_s in Coq as follows.

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

Details for this design will be discussed in section 2.4

2.2 Split result interface

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

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

Similar to Clight-A syntax C_c, the split result is also parameterized by a simpler syntax in implementation. For simple results, the assertion is treated simply as a placeholder and there are no logical variables involved. For full results, each result type is a Coq Inductive type with an extra constructor to introduce logical variables into assertions. A complete result element p is in the dependepent type on a simple result p_s if p and p_s have the same list of basic statements.

Again, we will show how this dependepent relation is useful in the splitting function implementation in Coq in Section 2.4.

2.3 Interpreting split result

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

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

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

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

$$
\begin{aligned}
\text{semax_pre}(P, \vec{c_b} \dashv \{Q\}) &= \{P\}\ \text{to_Cstm}(\vec{c_b})\ \left\{Q, [\vec{\top}]\right\}\\
\text{semax_atom}(P, Q, \vec{c_b}) &= \{P\}\ \text{to_Cstm}(\vec{c_b})\ \left\{Q, [\vec{\top}]\right\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{semax_post}(Q, \{P\} \vdash \vec{c_b}) && = \{P\}\ \text{to_Cstm}(\vec{c_b})\ \left\{Q, [\vec{\top}]\right\}\\
&\text{semax_post}(Q, \forall x.\ p_{\vdash}) && = \forall x.\ \text{semax_post}(Q, p_{\vdash})\\
&\text{semax_path}(\{P\} \vdash \vec{c_b} \dashv \{Q\}) && = \{P\}\ \text{to_Cstm}(\vec{c_b})\ \left\{Q, [\vec{\top}]\right\}\\
&\text{semax_path}(\forall x.\ p_{\vdash\dashv}) && = \forall x.\ \text{semax_path}(p_{\vdash\dashv})
\end{aligned}
$$

$$
\begin{aligned}
&\quad \text{semax_atom_ret}(P, Q, \vec{c_b}; ?e)\\
&= \{P\}\ \text{to_Cstm}(\vec{c_b}); \text{return } e?\ \{\top, [\top, \top, Q]\}\\
&\quad \text{semax_post_ret}(Q, \{P\} \vdash \vec{c_b}; ?e)\\
&= \{P\}\ \text{to_Cstm}(\vec{c_b}); \text{return } e?\ \{\top, [\top, \top, Q]\}\\
&\quad \text{semax_post_ret}(Q, \forall x.\ p^{\text{return}}_{\text{post}})\\
&= \forall x. \text{semax_post_ret}(Q, p^{\text{return}}_{\text{post}})
\end{aligned}
$$

$$
\begin{aligned}
&\quad \text{semax_split}(P, Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}},\\
&\qquad \left\{\begin{array}{l}
\boldsymbol{p}^{\text{nor}}_{-}, \boldsymbol{p}^{\text{brk}}_{-}, \boldsymbol{p}^{\text{con}}_{-}, \boldsymbol{p}^{\text{ret}}_{-},\\
\boldsymbol{p}^{\text{nor}}_{\vdash}, \boldsymbol{p}^{\text{brk}}_{\vdash}, \boldsymbol{p}^{\text{con}}_{\vdash}, \boldsymbol{p}^{\text{ret}}_{\vdash},\\
\boldsymbol{p}_{\dashv}, \boldsymbol{p}_{\vdash\dashv}
\end{array}\right\})\\
&= \text{Forall (semax_atom } P\ Q)\ \boldsymbol{p}^{\text{nor}}_{-}\\
&\wedge \text{Forall (semax_atom } P\ Q_{\text{brk}})\ \boldsymbol{p}^{\text{brk}}_{-}\\
&\wedge \text{Forall (semax_atom } P\ Q_{\text{con}})\ \boldsymbol{p}^{\text{con}}_{-}\\
&\wedge \text{Forall (semax_atom_ret } P\ Q_{\text{ret}})\ \boldsymbol{p}^{\text{ret}}_{-}\\
&\wedge \text{Forall (semax_post } Q)\ \boldsymbol{p}^{\text{nor}}_{\vdash}\\
&\wedge \text{Forall (semax_post } Q_{\text{brk}})\ \boldsymbol{p}^{\text{brk}}_{\vdash}\\
&\wedge \text{Forall (semax_post } Q_{\text{con}})\ \boldsymbol{p}^{\text{con}}_{\vdash}\\
&\wedge \text{Forall (semax_post_ret } Q_{\text{ret}})\ \boldsymbol{p}^{\text{ret}}_{\vdash}\\
&\wedge \text{Forall (semax_pre } P)\ \boldsymbol{p}_{\dashv}\\
&\wedge \text{Forall (semax_path)}\ \boldsymbol{p}_{\vdash\dashv}
\end{aligned}
$$

Figure 4. semantics of split results

To reuse the verification infrastructures, we choose to reinterpret the split result into the Hoare triples defined by VST. We implement the semantics interpretation procedure as Coq functions in Figure 4.

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

path in question only cares about the case where e evaluates to true.

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

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

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

2.4 The split function

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

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

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

For splitting $c_1; c_2$, we first split c_1 and c_2 into paths $\boldsymbol{p}$'s and $\boldsymbol{q}$'s[3], and then construct the split result of $c_1; c_2$ by connecting corresponding control flow paths in $\boldsymbol{p}$'s and $\boldsymbol{q}$'s. For example, the break post-partial paths have three components, the ones from splitting c_1 ($\boldsymbol{p}_{\vdash}^{\mathrm{brk}}$), the ones from splitting c_2 ($\boldsymbol{q}_{\vdash}^{\mathrm{brk}}$), and the sequencing of the normal post-partial paths in c_1 and the break atom paths in c_2 ($\boldsymbol{p}_{\vdash}^{\mathrm{nor}} \cdot \boldsymbol{q}_{-}^{\mathrm{brk}}$).

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

The split result of loop (c_2) c_1 is more complicated. We need to restructure the control flow results related to break and continue in the loop body. First, since all break or continue related paths in the loop body will be digested by the loop structure, in the resulting split result, there are no (post-partial or atom) paths that exit with break or continue statuses. Next, the break post-partials ($\boldsymbol{p}_{\vdash}^{\mathrm{brk}}$, $\boldsymbol{q}_{\vdash}^{\mathrm{brk}}$) in the two loop bodies become normal post-partials when viewed outside the loop. The break atom paths ($\boldsymbol{p}_{-}^{\mathrm{brk}}$, $\boldsymbol{q}_{-}^{\mathrm{brk}}$) in the two loop bodies will be combined with all possible pre-conditions in the loop and also become part of normal post-partial paths and normal atom paths. The case for return related post-partial/atom paths works in a similar way as the case for break. For continue paths, those in the main loop body c_1 are processed similarly as normal paths - they will both execute the incremental loop body c_2 next, while for continue paths in the incremental loop body c_2, they are disallowed to be executed, so a false post-condition $\perp$ is added to continue post-partial/atom paths.

Dealing with logical variables. When splitting new existential logical variables with an ExGiven $x : A,\ \{P(x)\}\ c'$ constructor, the new variable will be treated differently in pre-partial paths and post-partial paths. For pre-partial paths, since program assertions that appear before the ex-given structure will not mention the new variable x, it is safe to move x into the assertion P as an existential quantifier, leaving the pre-partial paths to be a singleton of $[] \dashv \{\exists x : A.\ P(x)\}$. There are no atom paths in the split result of ex-given structures, since the strucutre itself ensure that a pre-condition $P(x)$ appear in the head of the program. Post-partial paths for each exit status have two components, the post-partial paths of the inner statement c' (where x in c will be quantified universally), and the pre-condition $P(x)$ sequenced with atom-paths that exit with the corresponding status. For full paths, in addition to the closed full paths that are already

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

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

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

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

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

$$
\begin{array}{ll}
\text{split (loop } (c_2)\ c_1) = & \text{if } (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{nor}} \neq \emptyset \text{ then Error} \\
& \text{else} \left\{ \begin{array}{l}
\boldsymbol{p}_{\vdash}^{\text{brk}} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{brk}} +\!\!+\, (\boldsymbol{p}_{\vdash}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{\vdash}^{\text{con}} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{nor}} \cdot (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{brk}} +\!\!+\, (\boldsymbol{q}_{\vdash}^{\text{nor}} +\!\!+\, (\boldsymbol{p}_{\vdash}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{\vdash}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{nor}}) \cdot \boldsymbol{p}_{-}^{\text{brk}}, \\
\emptyset, \\
\emptyset, \\
\boldsymbol{p}_{\vdash}^{\text{ret}} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{ret}} +\!\!+\, (\boldsymbol{p}_{\vdash}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{\vdash}^{\text{con}} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{nor}} \cdot (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}})) \cdot \boldsymbol{q}_{-}^{\text{ret}} +\!\!+\, (\boldsymbol{q}_{\vdash}^{\text{nor}} +\!\!+\, (\boldsymbol{p}_{\vdash}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{\vdash}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{nor}}) \cdot \boldsymbol{p}_{-}^{\text{ret}}, \\
\boldsymbol{p}_{-}^{\text{brk}} +\!\!+\, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{brk}}, \\
\emptyset, \\
\emptyset, \\
\boldsymbol{p}_{-}^{\text{ret}} +\!\!+\, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{ret}}, \\
\boldsymbol{p}_{\dashv} +\!\!+\, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}}) \cdot (\boldsymbol{q}_{\dashv} +\!\!+\, \boldsymbol{q}_{-}^{\text{con}} \cdot \{[\,]{-}\{\bot\}\}), \\
\boldsymbol{p}_{\vdash\dashv} +\!\!+\, \boldsymbol{q}_{\vdash\dashv} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{nor}} \cdot \boldsymbol{p}_{\dashv} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{con}} \cdot \{[\,]{-}\{\bot\}\}) \\
\quad +\!\!+\ (\boldsymbol{p}_{\vdash}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{\vdash}^{\text{con}} +\!\!+\, \boldsymbol{q}_{\vdash}^{\text{nor}} \cdot (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\, \boldsymbol{p}_{-}^{\text{con}})) \cdot (\boldsymbol{q}_{\dashv} +\!\!+\, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{\dashv} +\!\!+\, \boldsymbol{q}_{-}^{\text{con}} \cdot \{[\,]{-}\{\bot\}\})
\end{array} \right\}
\end{array}
$$

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

Figure 5. Split function

split and collected in c', we also need to sequence the precondition $P(x)$ to the pre-partial paths in c'. Note that this connecting operation requires unifying the existential variable x with the one in pre-partial paths split from c, so that the logical variable x can be shared among the pre-/post-conditions of each combined full path.

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

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

.

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

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

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

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

```
Inductive S_statement : Type :=
| Sassert
| Ssequence (c1 c2 : S_statement)
| ...
.

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

Inductive S_result : Type := ... .

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

Fixpoint S_split (s: S_statement) : S_result.

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

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

2.5 Soundness

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

Theorem 2.1 (Soundness)**.** *For any Clight-A program p, precondition P and post-conditions $\vec{Q}$, if* split $p \neq$ *Error and semax_split*$(P, \vec{Q}, \text{split } p)$ *holds, then Hoare triple* $\{P\}\ p\ \left\{\vec{Q}\right\}$ *holds.* [4].

The soundness theorem is proved by induction on the statement to be split. Almost all of the soundness proof can be done on a logical level except one lemma (the conjunction rule, which will be proved in Section 3). The Hoare logic rules we uses are presented in Figure 6 and 8. We use $\Sigma; \Gamma$ to represent the proof context of logical variables and pure propositions that can be used to derive Hoare triples in the logical rules, but we will omit them in following text since they stays the same or the change of proof context is easy to be recognized.

The rules are formalized in a deeply embedded fashion, i.e. as an Inductive relation in Coq. The benefit of deeply embedded Hoare logic formalization is that it makes a rich bundle of derived rules readily available, a few of which are shown in Figure 7. We also make use of the following inversion lemmas when proving the soundness of our splitting function.

Lemma 2.2 (Inversion on sequencing)**.** *If* $\{P\}\ c_1; c_2\ \left\{Q, [\vec{Q'}]\right\}$, *then* $\{P'\}\ c_1\ \left\{\exists R : \textit{assert}.\ R \wedge \{R\}\ c_2\ \left\{Q, [\vec{Q'}]\right\}, [\vec{Q'}]\right\}$.

Lemma 2.3 (Inversion on if-branching)**.**
If $\{P\}\ \text{if } (b)\ c_1 \text{ else } c_2\ \left\{Q, [\vec{Q'}]\right\}$, *then*

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

Lemma 2.4 (Inversion on skip)**.**
If $\{P\}\ \text{skip}\ \left\{Q, [\vec{Q'}]\right\}$, *then* $P \vDash Q$

Lemma 2.5 (Inversion on break)**.**
If $\{P\}\ \text{break}\ \{Q, [Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}]\}$, *then* $P \vDash Q_{\text{brk}}$

The split results of basic operations are sound by definition. In fact, during the proof, we do not even need to look at the detailed rules for primary statements, since we choose to interpret the split result in the same way as the original statement. For control flow statements such as break, applying the inversion Lemma 2.4 to the corresponding control flow path can complete the proof. The rest of this section will focus on proving the soundness for compound statements.

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

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

$$\textsc{Semax-Skip}\ \frac{}{\Sigma;\Gamma \vdash \{P\}\ \mathsf{skip}\ \left\{P, [\vec{Q}]\right\}} \qquad \textsc{Semax-Break}\ \frac{}{\Sigma;\Gamma \vdash \{Q_{\mathrm{brk}}\}\ \mathsf{break}\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}}$$

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

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

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

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

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

Figure 6. Proof rules of C Hoare logic

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

$$\textsc{Seq-Assoc}\ \frac{\Sigma;\Gamma \vdash \{P\}\ c_1; (c_2; c_3)\ \left\{Q, [\vec{Q'}]\right\}}{\Sigma;\Gamma \vdash \{P\}\ (c_1; c_2); c_3\ \left\{Q, [\vec{Q'}]\right\}} \qquad \textsc{NoContinue}\ \frac{c\ \text{contains no continue} \qquad \Sigma;\Gamma \vdash \{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}, Q_{\mathrm{ret}}]\}}{\Sigma;\Gamma \vdash \{P\}\ c\ \{Q, [Q_{\mathrm{brk}}, Q_{\mathrm{con}}', Q_{\mathrm{ret}}]\}}$$

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

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

Figure 7. Derived rules from C Hoare logic

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

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

Lemma 2.6. *For any program c and post-condition $\vec{Q}$, it holds that* $\{\exists P : assert.\ P \wedge \{P\}\ c\ \left\{\vec{Q}\right\}\}\ c\ \left\{\vec{Q}\right\}$.

If we take a closer look at Lemma 2.2, it is essentially stating that the weakest pre-condition of the second statement can be used to serve as the intermediate assertion for sequential composition. Based on Lemma 2.2, we can prove for each type of paths in the split results a corresponding inversion lemma on the · operator.

Proposition 2.7 (Inversion lemmas for split results)**.**

1. *If* $semax_atom(P, Q, p_- \cdot q_-)$*, then* $semax_atom(P, \exists R.\ R \wedge semax_atom(R, Q, q_-), p_-)$
2. *If* $semax_atom_ret(P, Q, p_- \cdot q_-^{ret})$*, then* $semax_atom(P, \exists R.\ R \wedge semax_atom_ret(R, Q, q_-^{ret}), p_-)$
3. *If* $semax_pre(P, p_- \cdot q_\dashv)$*, then* $semax_atom(P, \exists R.\ R \wedge semax_pre(R, q_\dashv), p_-)$
4. *If* $semax_post(Q, p_\vdash \cdot q_-)$*, then* $semax_post(Q, \exists R.\ R \wedge semax_atom(R, Q, q_-), p_\vdash)$
5. *If* $semax_post_ret(Q, p_\vdash \cdot q_-^{ret})$*, then* $semax_post(Q, \exists R.\ R \wedge semax_atom_ret(R, Q, q_-^{ret}), p_\vdash)$
6. *If* $semax_path(p_\vdash \cdot q_\dashv)$*, then* $semax_post(\exists R.\ R \wedge semax_pre(R, q_\dashv), p_\vdash)$

The inversion lemma also holds for the case where the first argument of · is a post-partial path with quantified structures. Consider item 4 in Proposition 2.7, for example, the proof can be done by induction on $p_\vdash$. When $p_\vdash = \forall x.\ p'_\vdash$, we know that $\forall x.\text{semax_post}(Q, p'_\vdash \cdot q_-)$. The goal can be simplified as $\forall x.\text{semax_post}(Q, \exists R.\ R \wedge \text{semax_atom}(R, Q, q_-), p'_\vdash)$, which can be proved directly from the induction hypothesis.

Combining weakest pre-conditions. With Proposition 2.7, we can collect the intermediate assertions for each control flow that crosses the two sub-statements, but to apply the logical rules for the original Clight program, we need to combine all the intermediate assertions into one.

We start with combining intermediate assertions for the same type of paths in the split result. Note that the · operator works on a set of paths instead of individual path in the split function, so Proposition 2.7 should be extended to the following form [5]

Proposition 2.8 (Grouped Inversion lemmas)**.**

1. *If* $\boldsymbol{q}_- \neq \emptyset$ *and* $semax_atom(P, Q, \boldsymbol{p}_- \cdot \boldsymbol{q}_-)$*, then* $semax_atom(P, \exists R.\ R \wedge semax_atom(R, Q, \boldsymbol{q}_-), \boldsymbol{p}_-)$
2. *If* $\boldsymbol{q}_-^{ret} \neq \emptyset$ *and* $semax_atom_ret(P, Q, \boldsymbol{p}_- \cdot \boldsymbol{q}_-^{ret})$*, then* $semax_atom(P, \exists R.\ R \wedge semax_atom_ret(R, Q, \boldsymbol{q}_-^{ret}), \boldsymbol{p}_-)$
3. *If* $\boldsymbol{q}_\dashv \neq \emptyset$ *and* $semax_pre(P, \boldsymbol{p}_- \cdot \boldsymbol{q}_\dashv)$*, then* $semax_atom(P, \exists R.\ R \wedge semax_pre(R, \boldsymbol{q}_\dashv), \boldsymbol{p}_-)$
4. *If* $\boldsymbol{q}_- \neq \emptyset$ *and* $semax_post(Q, \boldsymbol{p}_\vdash \cdot \boldsymbol{q}_-)$*, then* $semax_post(Q, \exists R.\ R \wedge semax_atom(R, Q, \boldsymbol{q}_-), \boldsymbol{p}_\vdash)$
5. *If* $\boldsymbol{q}_-^{ret} \neq \emptyset$ *and* $semax_post_ret(Q, \boldsymbol{p}_\vdash \cdot \boldsymbol{q}_-^{ret})$*, then* $semax_post(Q, \exists R.\ R \wedge semax_atom_ret(R, Q, \boldsymbol{q}_-^{ret}), \boldsymbol{p}_\vdash)$
6. *If* $\boldsymbol{q}_\dashv \neq \emptyset$ *and* $semax_path(\boldsymbol{p}_\vdash \cdot \boldsymbol{q}_\dashv)$*, then* $semax_post(\exists R.\ R \wedge semax_pre(R, \boldsymbol{q}_\dashv), \boldsymbol{p}_\vdash)$

Proposition 2.8 can be proved by induction fisrt on the size of set on the LHS of the · operator then on the size of set on the RHS. The induction on the LHS argument is straightforward, since the intermediate assertions we obtain from the inversion lemmas are the same if we fix the RHS argument. Consider the case for the second induction when proving Proposition 2.8 (4). Assume $\boldsymbol{q}_- = q_- :: \boldsymbol{q'}_-$. Applying Proposition 2.7 (4) to q_- we get

$$\text{semax_post}(\exists R_1.\ \text{semax_pre}(R_1, q_-), \boldsymbol{p}_\vdash)$$

. The induction hypothesis gives

$$\text{semax_post}(\exists R_2.\ \text{semax_pre}(R_2, \boldsymbol{q'}_-), \boldsymbol{p}_\vdash)$$

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

$$\begin{aligned} & \exists R_1.\ \text{semax_pre}(R_1, q_-) \wedge \exists R_2.\ \text{semax_pre}(R_2, \boldsymbol{q'}_-) \\ \vDash\ & \exists R.\ \text{semax_pre}(R, \boldsymbol{q}_-) \end{aligned}$$

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

Proposition 2.9 (Conjunction rule on paths)**.**

1. *If* $semax_post(Q_1, q_\vdash)$ *and* $semax_post(Q_2, q_\vdash)$*, then* $semax_post(Q_1 \wedge Q_2, q_\vdash)$
2. *If* $semax_atom(P, Q_1, q_-)$ *and* $semax_atom(P, Q_2, q_-)$*, then* $semax_atom(P, Q_1 \wedge Q_2, q_-)$
3. *If* $semax_atom_ret(P, Q_1, q_-^{ret})$ *and* $semax_atom_ret(P, Q_2, q_-^{ret})$*, then* $semax_atom_ret(P, Q_1 \wedge Q_2, q_-^{ret})$
4. *If* $semax_post(Q_1, \boldsymbol{q}_\vdash)$ *and* $semax_post(Q_2, \boldsymbol{q}_\vdash)$*, then* $semax_post(Q_1 \wedge Q_2, \boldsymbol{q}_\vdash)$
5. *If* $semax_atom(P, Q_1, \boldsymbol{p}_-)$ *and* $semax_atom(P, Q_2, \boldsymbol{p}_-)$*, then* $semax_atom(P, Q_1 \wedge Q_2, \boldsymbol{p}_-)$
6. *If* $semax_atom_ret(P, Q_1, \boldsymbol{p}_-^{ret})$ *and* $semax_atom_ret(P, Q_2, \boldsymbol{p}_-^{ret})$*, then* $semax_atom_ret(P, Q_1 \wedge Q_2, \boldsymbol{p}_-^{ret})$

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

Theorem 2.10 (Conjunction Rule). *if Hoare triples* $\{P\}\ c\ \left\{Q_1, [\vec{Q'_1}]\right\}$ *and* $\{P\}\ c\ \left\{Q_1, [\vec{Q'_2}]\right\}$ *are derivable, then* $\{P\}\ c\ \left\{Q_1 \wedge Q_2, [\vec{Q'_1} \wedge \vec{Q'_2}]\right\}$ *is derivable holds.*

However, the conjunction rule is not ubiquitous among Hoare logic variants proposed in literatures. For example, the current VST program logic cannot derive the conjunction rule. We will leave the discussion of the conjunction rule to Section 3. For now, we assume that the conjunction rule is available, so that we construct the intermediate assertion for Semax-Seq as follows:

[5] We will abuse notations like $\text{semax_path}(\boldsymbol{p}_\mapsto)$ where $\boldsymbol{p}_\mapsto$ is a set of paths to represent (Forall semax_path $\boldsymbol{p}_\mapsto$) in the following report.

$$R = \left\{ \begin{array}{l} \exists R.\ R \wedge \text{semax_pre}(R, \boldsymbol{q}_{\dashv}) \\ \quad \wedge \text{semax_atom}(R, Q, \boldsymbol{q}_{-}^{\text{nor}}) \\ \quad \wedge \text{semax_atom}(R, Q_{\text{brk}}, \boldsymbol{q}_{-}^{\text{brk}}) \\ \quad \wedge \text{semax_atom}(R, Q_{\text{con}}, \boldsymbol{q}_{-}^{\text{con}}) \\ \quad \wedge \text{semax_atom_ret}(R, Q_{\text{ret}}, \boldsymbol{q}_{-}^{\text{ret}}) \end{array} \right\}$$

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

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

$$I = \left\{ \begin{array}{l} \exists R.\ R \wedge \text{semax_pre}(R, \boldsymbol{p}_{\dashv}) \\ \quad \wedge \text{semax_pre}(R, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{\dashv}) \\ \quad \wedge \text{semax_atom}(R, \perp, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{con}}) \\ \quad \wedge \text{semax_atom}(R, Q, \boldsymbol{p}_{-}^{\text{brk}}) \\ \quad \wedge \text{semax_atom}(R, Q, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{brk}}) \\ \quad \wedge \text{semax_atom_ret}(R, Q_{\text{ret}}, \boldsymbol{p}_{-}^{\text{ret}}) \\ \quad \wedge \text{semax_atom_ret}(R, Q_{\text{ret}}, (\boldsymbol{p}_{-}^{\text{nor}} +\!\!+\ \boldsymbol{p}_{-}^{\text{con}}) \cdot \boldsymbol{q}_{-}^{\text{ret}}) \end{array} \right\}$$

$$I_{\text{con}} = \left\{ \begin{array}{l} \exists R.\ R \wedge \text{semax_pre}(R, \boldsymbol{q}_{\dashv}) \\ \quad \wedge \text{semax_pre}(R, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{\dashv}) \\ \quad \wedge \text{semax_atom}(R, \perp, \boldsymbol{q}_{-}^{\text{con}}) \\ \quad \wedge \text{semax_atom}(R, Q, \boldsymbol{q}_{-}^{\text{brk}}) \\ \quad \wedge \text{semax_atom}(R, Q, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{-}^{\text{brk}}) \\ \quad \wedge \text{semax_atom_ret}(R, Q_{\text{ret}}, \boldsymbol{q}_{-}^{\text{ret}}) \\ \quad \wedge \text{semax_atom_ret}(R, Q_{\text{ret}}, \boldsymbol{q}_{-}^{\text{nor}} \cdot \boldsymbol{p}_{-}^{\text{ret}}) \end{array} \right\}$$

By inversion from the premise that

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

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

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

Recall the interpretation function we have defined for conditional expression:

$$\text{to_Cstm}([e]) = \text{if } (e) \text{ skip else break}$$

. To translate from them, we make use of the inversion lemmas that are available in deep-embedded VST program logic. The following propositions can be derived from the Lemma 2.3, 2.4 and 2.5.

Proposition 2.11 (Conditional Expression lemmas)**.**

1. *If* $\textit{semax_pre}(P, \{[e]\} \cdot \boldsymbol{p}_{\dashv})$*, then* $\textit{semax_pre}(P \wedge [\![e]\!] = \text{true}, \boldsymbol{p}_{\dashv})$
2. *If* $\textit{semax_atom}(P, Q, \{[e]\} \cdot \boldsymbol{p}_{-})$*, then* $\textit{semax_atom}(P \wedge [\![e]\!] = \text{true}, Q, \boldsymbol{p}_{-})$
3. *If* $\textit{semax_atom_ret}(P, Q, \{[e]\} \cdot \boldsymbol{p}_{-}^{ret})$*, then* $\textit{semax_atom_ret}(P \wedge [\![e]\!] = \text{true}, Q, \boldsymbol{p}_{-}^{ret})$

We show the proof detail for item 1.

Proof. We consider any pre-partial path $p_{\dashv}$ in $\boldsymbol{p}_{\dashv}$. From Proposition 2.7 (3), we have that

$$\{P\}\ \text{to_Cstm}([e])\ \left\{\exists R.\ R \wedge \text{semax_pre}(R, p_{\dashv}), [\vec{\perp}]\right\}$$

Applying lemmas 2.3, 2.4, and 2.5, we obtain

$$\begin{array}{l} P \vDash \exists P' : \text{assert},\ P' \\ \qquad \wedge (P' \wedge [\![e]\!] = \text{true} \vDash \exists R.\ R \wedge \text{semax_pre}(R, p_{\dashv})) \\ \qquad \wedge (P' \wedge [\![e]\!] = \text{false} \vDash \perp) \end{array}$$

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

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

which follows directly from the fact that

$$\text{semax_pre}(\exists R.\ \text{semax_pre}(R, p_{\dashv})), p_{\dashv})$$

□

After transformation using Proposition 2.11 in the premise of the soundness theorem, the induction hypothesis can be directly applied to complete the proof. We omit the dual case where the $\neg e$ branch is taken.

Soundness of ex-given structure. The premise that

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

leads to the following two propositions.

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

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

3 Conjunction rule

The key missing ingredient of the soundness proof in the previous section is the conjunction rule (Theorem 2.10).

3.1 Proving the conjunction rule in VST

The current VST program logic, as an impredicative higher order concurrent separation logic, does not imply the conjunction rule. To make the conjunction rule available in VST-A, the logical rules we have presented in Figure 6 and 8 is actually a stronger (but still sound) variant of the current VST program logic. To be specific, we remove logical operators related to concurrency, and add extra constraints to SEMAX-CALL.

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

Theorem 3.1 (Conjunction Rule). *if Hoare triples* $\{P\}\ c\ \left\{Q_1, [\vec{Q'_1}]\right\}$ *and* $\{P\}\ c\ \left\{Q_1, [\vec{Q'_2}]\right\}$ *are derivable, then* $\{P\}\ c\ \left\{Q_1 \wedge Q_2, [\vec{Q'_1} \wedge \vec{Q'_2}]\right\}$ *is derivable.*

Proof. We prove by induction on the statement c.

- c = skip. Applying Lemma 2.4 on premises we get $P \vDash Q_1$ and $P \vDash Q_2$. Thus, $P \vDash Q_1 \wedge Q_2$ is derivable, and the result follows from SEMAX-SKIP. The proof is simiar when c is break, continue, and return?e.
- $c = c_1; c_2$. Applying Lemma 2.2 on premises we get

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

  We can apply the induction hypothesis of c_1 and make use of SEMAX-CONSEQ to obtain

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

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

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

  Using EXTRACT-EXISTS and EXTRACT-PROP we can extract R and the two Hoare triples into the proof context. By applying the induction hypothesis of c_2 on the two Hoare triples, we can obtain the result.
- c = if (e) c_1 else c_2. By Lemma 2.3, we can obtain R_1 and R_2 and the following triples in the proof context:

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

  Both R_1 and R_2 are derivable from P, and the result follows by applying the induction hypothesis on respective triples.
- c = loop (c_2) c_1. The inversion lemma for loop structures will provide two pair of loop invariants I_1, I_1^{incr} and I_2, I_2^{incr} for the two premises respectively, where I_1 and I_2 are derivable from P, together with the following triples available:

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

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

□

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

3.2 Conjunction rule for memory load

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

The logical rules for primary statements have been presented in Figure 8. We begin with rules related to assignment statements $e_1 := e_2$.

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

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

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

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

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

Figure 8. Proof rules of primary C assignment commands

checking verification. We will abuse the notation of mapsto($\mapsto$), and omit the type-checking parameter in this report.

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

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

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

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

Lemma 3.2 (Inversion on Sset). *If* $\{P\}\ x := e\ \{Q, [\vec{Q'}]\}$, *then*

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

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

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

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

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

Proposition 3.3 (Cross Split Property). *A join relation for permission-shares has the cross split property if* $\pi = \pi_a \oplus \pi_b \wedge \pi = \pi_c \oplus \pi_d \Rightarrow \exists \pi_{ac}\ \pi_{ad}\ \pi_{bc}\ \pi_{bd}, \pi_{ac} \oplus \pi_{ad} = \pi_a \wedge \pi_{bc} \oplus \pi_{bd} = \pi_b \wedge \pi_{ac} \oplus \pi_{bc} = \pi_c \wedge \pi_{ad} \oplus \pi_{bd} = \pi_d$.

Proposition 3.4 (Cancellative Property). *A join relation for permission-shares has the cancellative property if* $\pi = \pi_a \oplus \pi_b \wedge \pi = \pi_{a'} \oplus \pi_b \Rightarrow \pi_a = \pi_{a'}$.

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

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

Theorem 3.5. *If* π_1 *and* π_2 *are readable shares, then*

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

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

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

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

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

3.3 Preciseness of memory write

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

Lemma 3.6 (Inversion on Sassign)**.** *If* $\{P\}\ e_1 := e_2\ \left\{Q, [\vec{Q'}]\right\}$, *then*

$$P \models \exists \pi\ p\ v\ v'.\ \pi \text{ is writable share} \wedge \triangleright \begin{pmatrix} \&e_1 \Downarrow p \wedge e_2 \Downarrow v' \wedge \\ (p \mapsto_\pi v * (p \mapsto_\pi v' -\!\!* Q)) \end{pmatrix}$$

After applying Lemma 3.6 on premises of the conjunction rule, the proof obligation left is as follows, which also requires model-level proving.

Theorem 3.7. *If* π_1 *and* π_2 *are writable shares, then*

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

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

Applying Theorem 3.5, we can show that $v_1 = v_2$ and that there exists a splitting for $r = r_{\mapsto}^{\pi_1 \cup \pi_2} \oplus r_{\text{rem}}^{\pi_1 \cup \pi_2}$ where $r_{\mapsto}^{\pi_1 \cup \pi_2} \models p \mapsto_{\pi_1 \cup \pi_2} v_1$.

We are left to prove that given any model $r_{\mapsto}'$ that satisfies $p \mapsto_{\pi_1 \cup \pi_2} v_1$, $r_{\mapsto}' \oplus r_{\text{rem}}^{\pi_1 \cup \pi_2}$ satisfies $P_1 \wedge P_2$. Without loss of generality, we show that $r_{\mapsto}' \oplus r_{\text{rem}}^{\pi_1 \cup \pi_2} = r' \models P_1$. Figure 9 plots the layout of the join relations that have been introduced so far in the proof.

To prove a model satisfying P_1, we must make use of the fact that $r_{\text{rem}}^{\pi_1} \models p \mapsto_{\pi_1} v' -\!\!* P_1$ The only way to relate r' with $r_{\text{rem}}^{\pi_1}$ is through $r_{\text{rem}}^{\pi_1 \cup \pi_2}$, the common sub-model of

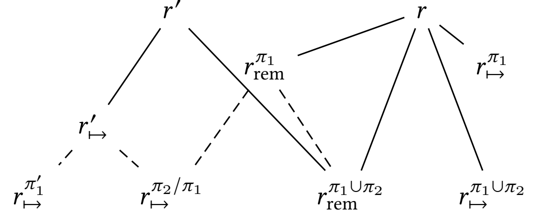


Figure 9. Semantic model layout for the proof of Theorem 3.7. (Dotted lines indicate the join relation to be proved)

r' and r. The idea is to "borrow" a sub-model from $r_{\mapsto}'$, to supplement the model $r_{\text{rem}}^{\pi_1 \cup \pi_2}$, so that the supplement can be matched with $r_{\text{rem}}^{\pi_1}$.

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

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

, where $[r@l/\pi]$ indicates a resource on location l that has the same value as $r@l$ but with the permissions reassigned as π. The two dotted join relations in Figure 9 can be verified.

1. $r_{\mapsto}' = r_{\mapsto}^{\pi_1'} \oplus r_{\mapsto}^{\pi_2/\pi_1}$ follows directly from the fact that $\pi_1 \cup \pi_2 = \pi_1 \oplus (\pi_2/\pi_1)$.
2. $r_{\text{rem}}^{\pi_1} = r_{\mapsto}^{\pi_2/\pi_1} \oplus r_{\text{rem}}^{\pi_1 \cup \pi_2}$ requires inversion on $r = r_{\text{rem}}^{\pi_1} \oplus r_{\mapsto}^{\pi_1}$ and $r = r_{\text{rem}}^{\pi_1 \cup \pi_2} \oplus r_{\mapsto}^{\pi_1 \cup \pi_2}$.
   For l referenced by p, let the permissions for $r@l$, $r_{\text{rem}}^{\pi_1}@l$ and $r_{\text{rem}}^{\pi_1 \cup \pi_2}@l$ be π, π_3 and π_4 respectively. Then, we have $\pi = \pi_3 \oplus \pi_1$ and $\pi = \pi_4 \oplus (\pi_1 \cup \pi_2)$. To prove $r_{\text{rem}}^{\pi_1}@l = r_{\mapsto}^{\pi_2/\pi_1}@l \oplus r_{\text{rem}}^{\pi_1 \cup \pi_2}@l$, it suffices to show that $\pi_3 = (\pi_2/\pi_1) \oplus \pi_4$. The cancellative property is used here.
   For l not referenced by p, no resources are defined on $r_{\mapsto}^{\pi_1}$ and $r_{\mapsto}^{\pi_1 \cup \pi_2}$, so $r_{\text{rem}}^{\pi_1}$ and $r_{\text{rem}}^{\pi_1 \cup \pi_2}$ are the same. Since we define no resources for $r_{\mapsto}^{\pi_2/\pi_1}$, the join relation holds.

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

3.4 Function call with precise specifications

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

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

Definition 3.8 (Function subsumption)**.** A function specification $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$ is subsumed by another function specification $[\vec{A'}]\{\lambda\vec{y}.P'\}\ \{\lambda r.Q'\}$ if

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

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

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

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

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

Definition 3.9 (Precise function specification)**.** A function specification $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$ is precise if for any formal parameters $\vec{b}$, return value a and assertions R_1, R_2, it holds that

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

The definition looks very similar to Theorem 9 that we have proved for the memory writing operation, which can serve as a justification for our definition.

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

Definition 3.10 (Precise predicate)**.** A predicate P is precise if for any Q_1, Q_2,

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

Clearly, if we remove the existential quantifier $\vec{A}$ from Definition 3.9, then all function specifications written with precise predicates are precise. Therefore, we consider our definition of precise function specification fits into a more general setting, allowing the instantiations of the function specification to be different, while still being able to derive the conjunction rule of function calls, as we will see below.

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

Lemma 3.11 (Inversion on function call)**.** *If* $\{S\}\ a := f(\vec{e})\ \left\{R, [\vec{R'}]\right\}$, *then*

$$\begin{aligned} S \vDash\ \ &\exists A\ (\lambda\vec{y}.P)\ (\lambda r.Q)\ \vec{b}. \\ &\Delta(f) <: [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} \wedge \\ &\Delta(f)\ \textit{is a precise specification} \wedge \\ &\triangleright \begin{pmatrix} \vec{e} \Downarrow \vec{b} \wedge \\ \exists \vec{x} : \vec{A}.\ P\,\vec{b}\,\vec{x} * (Q\,a\,\vec{x} \twoheadrightarrow R) \end{pmatrix} \end{aligned}$$

By applying Lemma 3.11, the proof obligation is left as follows.

Theorem 3.12. *If* $\Delta(f) = [\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\}$, $\Delta(f)$ *is a precise function specification and*

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

then

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

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

Lemma 3.13. *If* $[\vec{A}]\{\lambda\vec{y}.P\}\ \{\lambda r.Q\} <: [\vec{A'}]\{\lambda\vec{y}.P'\}\ \{\lambda r.Q'\}$, *then for any formal parameters* $\vec{b}$, *return value a and assertion R,*

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

Proof. By logical derivation,

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

□

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

Last Updated on May 2, 2022.