VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs

As Published: https://doi.org/10.1007/s10817-018-9457-5

Publisher: Springer Netherlands

Persistent URL: https://hdl.handle.net/1721.1/131755

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Noname manuscript No.
(will be inserted by the editor)

VST-Floyd: A separation logic tool to verify correctness of C programs

Qinxiang Cao · Lennart Beringer
Samuel Gruetter · Josiah Dodds
Andrew W. Appel

Received: date / Accepted: date

Abstract The Verified Software Toolchain builds foundational machine-checked proofs of the functional correctness of C programs. Its program logic, Verifiable C, is a shallowly embedded higher-order separation Hoare logic which is proved sound in Coq with respect to the operational semantics of CompCert C light. This paper introduces VST-Floyd, a verification assistant which offers a set of semiautomatic tactics helping users build functional correctness proofs for C programs using Verifiable C.

1 Introduction

In our interconnected world, software bugs can seriously compromise our safety and security. Protection mechanisms such as operating systems, crypto libraries, and language runtimes can protect buggy programs from each other—but those protection mechanisms are also software, often written in low-level programming languages such as C. To provide adequate safety or protection, these software components in C must be functionally correct.

To assure functional correctness of C programs, we can use Hoare logic [16] and its extensions such as separation logic [29]. But such correctness proofs are large and complex enough that we cannot trust them unless they are machine-checked. The Verified Software Toolchain (VST) is a set of verified tools that enable users to formally verify the functional correctness of C programs using Hoare logic: at bottom, the CompCert verified C compiler from INRIA; above that, *core Verifiable C*, a separation logic proved sound in Coq with respect to the operational semantics of CompCert C light; above that, *Verifiable C*, a derived separation logic that supports

Qinxiang Cao
Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540
E-mail: caoqinxiang@gmail.com

Andrew W. Appel
Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540
E-mail: appel@princeton.edu

proof automation; above that, a proof automation system to assist the user in applying the program logic to the program. The users' proofs, the soundness of the tools, and the correspondence of the compiled program to the formal model, are all machine checked.

In this article we explain VST-Floyd, which includes a proof-automation system for interactively constructing functional correctness proofs of C programs, along with the special-purpose derived Hoare rules and predicate operators of Verifiable C that support this proof automation. Appel *et al.* [3] describe an early prototype of VST-Floyd, but in this paper we describe many new techniques, results, and engineering (every feature described in sections 3–9 is either completely new or substantially improved) since that publication. We design a canonical form of assertion for automatic generation of strongest postconditions. We offer separation logic predicates to describe data of C aggregate types stored in memory. The whole system acts as a tactic library enabling our users to proform forward verification.

This system has been used to construct correctness proofs of several real-world C programs, including components of various cryptographic libraries (see §11). Our Coq development can be found online at,

https://github.com/PrincetonUniversity/VST/releases/tag/v1.9

1.1 Hoare logic

Hoare logic is a formal system for reasoning about program correctness. It uses assertions to describe programs' behavior: a precondition P, a program c and a postcondition Q form a Hoare triple $\{P\}\ c\ \{Q\}$. The triple means that for any initial state s and ending state t, if $s \vDash P$ ("s satisfies P"), then running c from s is safe (in particular, it cannot result in undefined behavior); and if c terminates in state t, then $t \vDash Q$. Our tools implement a Hoare logic of *partial correctness*, meaning that c may loop infinitely.

Hoare logic is compositional, i.e. a Hoare triple of a program can be proved by the Hoare triples of its components. For example, here is the rule for sequential composition:

$$\text{HOARE-SEQ}\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}$$

Because of this compositionality, the proof of a Hoare triple can be written as a decorated program. Figure 1 shows the decorated program and its proof tree.

Using assertions and partially decorated programs to illustrate program correctness has already been widely used in software development.

1.2 Machine checkable proofs

Our previous work, core Verifiable C [3, Part III], is a shallowly embedded higher-order separation Hoare logic formalized in Coq. Coq is an interactive, programmable proof assistant in the tradition of Edinburgh LCF: users build proofs interactively by applying tactics, transitioning from a proof goal to zero or more subgoals that imply the original proof goal. When all proof goals are solved, a **Qed** command checks the correctness of the user-constructed proof.

$\{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b\}$
`z = x;`
$\{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b \wedge [\![\mathsf{z}]\!] = a\}$
`x = y;`
$\{[\![\mathsf{x}]\!] = b \wedge [\![\mathsf{y}]\!] = b \wedge [\![\mathsf{z}]\!] = a\}$
`y = z;`
$\{[\![\mathsf{x}]\!] = b \wedge [\![\mathsf{y}]\!] = a \wedge [\![\mathsf{z}]\!] = a\}$

$$\dfrac{\dfrac{\{[\![\mathsf{x}]\!] = a\} \; \mathtt{z = x;} \; \{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b \wedge [\![\mathsf{z}]\!] = a\}}{\{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b\} \; \mathtt{z = x;} \; \{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b \wedge [\![\mathsf{z}]\!] = a\}} \quad \dfrac{\dfrac{\{[\![\mathsf{x}]\!] = a \wedge \ldots\} \; \mathtt{x = y} \; \{[\![\mathsf{x}]\!] = b \wedge \ldots\}}{\{[\![\mathsf{x}]\!] = a \wedge \ldots\} \; \mathtt{x = y;} \; \{[\![\mathsf{x}]\!] = b \wedge \ldots\}} \quad \dfrac{\{\ldots \wedge [\![\mathsf{y}]\!] = b \wedge \ldots\} \; \mathtt{y = z} \; \{\ldots \wedge [\![\mathsf{y}]\!] = a \wedge \ldots\}}{\{\ldots \wedge [\![\mathsf{y}]\!] = b \wedge \ldots\} \; \mathtt{y = z;} \; \{\ldots \wedge [\![\mathsf{y}]\!] = a \wedge \ldots\}}}{\{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b\} \; \mathtt{x = y; \; y = z;} \; \{[\![\mathsf{x}]\!] = b \wedge [\![\mathsf{y}]\!] = a \wedge [\![\mathsf{z}]\!] = a\}}$$

$$\{[\![\mathsf{x}]\!] = a \wedge [\![\mathsf{y}]\!] = b\} \; \mathtt{z = x; \; x = y; \; y = z;} \; \{[\![\mathsf{x}]\!] = b \wedge [\![\mathsf{y}]\!] = a \wedge [\![\mathsf{z}]\!] = a\}$$

Fig. 1: Decorated program and proof tree

Verifiable C is used to prove Hoare triples of C programs. First, the front-end of the CompCert C compiler [25] parses and translates the C program into an abstract syntax tree in *C light*, a variant of C in which (for example) side-effects have been factored out of subexpressions into separate commands. The user writes a *function specification* for each function, giving the function's precondition and postcondition. Then the Hoare triple of each C light function body is a theorem to be proved in Coq; this proof goal can be broken into Hoare triples of smaller program fragments (Coq subgoals) by applying structural Hoare rules (such as HOARE-SEQ) using Coq tactics in Verifiable C. Triples of atomic C commands (such as assignment statements) can be proved using Verifiable C's atomic Hoare rules. The **Qed** commands at the end check the correctness, i.e. proofs of Hoare triples by Verifiable C in Coq are all machine checked.

Moreover, Verifiable C is proved sound [3, Part VI] with respect to the operational semantics of C light, formalized in Coq. The back end of CompCert translates C light to assembly language, and is proved correct w.r.t. the operational semantics of C light and assembly. Thus, the trusted base of functional correctness proofs by Verifiable C contains only Coq's kernel (proof checker) and the formalized assembly-language semantics for a target machine such as x86, ARM, PowerPC, or RISC-V.

Verifiable C is an extremely expressive program logic: it is a higher-order impredicative concurrent separation logic. That means it can reason about function pointers, higher-order predicate quantification, data abstraction, object protocols, shared-memory concurrency [26], and pointer data structures with mutation—all with respect to specifications of functional correctness in a general purpose logic, namely Coq. No other mechanized program logic for C that we know of has this level of expressiveness and flexibility.

The basic principles of VST function specifications and the soundness of Verifiable C have been described elsewhere [3]. In this paper we focus on how to automate proofs by forward interactive symbolic execution.

1.3 VST-Floyd

Using Verifiable C *directly* to verify C programs would be quite inconvenient! There would be two main problems: (1) Coq's notion of tactic-directed backward proof, when applied to Hoare logic, would be like writing a proof tree (such as in Fig. 1) from the bottom up. This does not correspond well to the programmer's intuition of executing the program in statement order. (2) C's semantics has many subtleties and corner cases. Verifiable C's primitive proof rules have many side conditions to account for these issues. Satisfying all these conditions could be quite tedious, requiring long proofs in Coq.

In this article, we present VST-Floyd, a lemma and tactic library which solves these two problems and helps users build Hoare triples in Verifiable C.

The most important feature of VST-Floyd is its forward proof style. Fig. 2 demonstrates a tiny example of it. One step of forward verification reduces the proof goal in 2(d) to 2(e) and another step reduces 2(e) to 2(f). The process simulates how one might write a decorated program: from 2(a) to 2(b) and from 2(b) to 2(c). During this process, the shaded lines in 2(a-c) represent Hoare triples which are not yet proved.

The VST-Floyd user does not see the decorated program directly, as in Fig. 2(a–c). Instead, one sees the corresponding proof goals in 2(d-f).

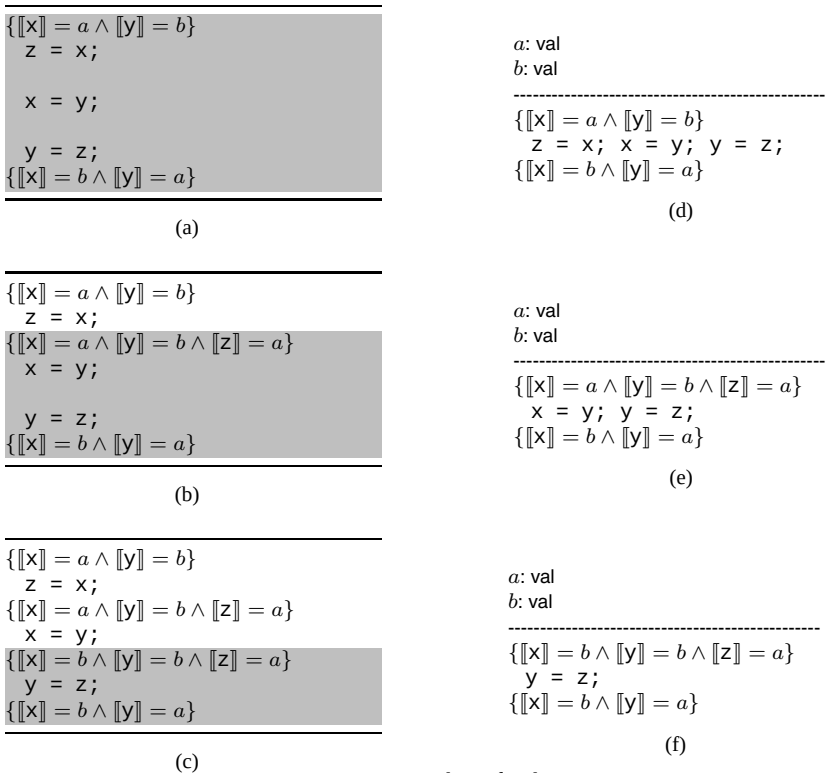


Fig. 2: Forward proof style

VST-Floyd helps automate forward verification. In the example above, users need not write the intermediate assertions in Fig. 2. Previous techniques for generating strongest postconditions [13] introduced an existential to bind the "old" value of a local variable. Our *canonical form* makes this simpler by making the old value explicit in the precondition, so that no existential is introduced. We automatically generate the strongest postconditions of atomic commands—without existentials—along with proof terms which certify the corresponding Hoare triples.

To enhance the usability of our verification tool, we provide a separation logic predicate $p \underset{t}{\longmapsto} v$ for writing concise assertions. This predicate says, data v of C type t is stored at address p. Here, t can be any C type including integers, floats, pointers, `struct`, `union`, and arrays; `struct`, `union`, and arrays can be nested.

Also, we provide useful tactics to manipulate Hoare triples in VST-Floyd. Rather than pushing symbolic execution (akin to strongest-postcondition generation) all the way through a program and then dealing with an intractable proof goal at the end, one can use the rule of consequence to adjust the assertions between commands:

$$\text{HOARE-PRE}\ \frac{P \vdash P' \qquad \{P'\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}}$$

The side condition $P \vdash P'$ is a separation logic entailment. VST-Floyd provides tactics to simplify (and sometimes even solve) entailments. Users may apply domain-specific theorems in these entailment proofs. Besides, VST-Floyd also provides tactics to manipulate quantifiers, separation logic subformulas, etc., in proof goals.

We have engineered Floyd to be *efficient.* Earlier versions of our tactics were so slow that it was impractical to verify real programs, so we learned how to reformulate our Hoare lemmas and tactics for faster verification.

We organize the rest of this article as follows. §2 briefly summarizes Verifiable C, our previous work. §3 describes VST-Floyd's data-structure assertions; §4 introduces the canonical form of our Floyd assertion language. §5 shows how to generate strongest postconditions of atomic commands. §6 and §7 introduce the interface and implementation of the tactics that perform forward verification. §8 introduces the tactics to perform structural proof rules. §9 presents tactics to manipulate and solve separation logic entailments. §10 and §11 describe the use of VST-Floyd in practice.

2 Background: Verifiable C

Verifiable C is a separation logic proved sound with respect to the operational semantics of CompCert C light, which is an early-stage intermediate language of the CompCert verified optimizing C compiler. CompCert offers a program called "clightgen" to generate corresponding C light programs from a C program.

The syntax of C light is very similar to C. The key differences that matter in this paper are that C light (1) distinguishes addressable variables and nonaddressable variables, (2) unifies different loop commands, and (3) has no side effects nested inside subexpressions.

In the C language, one can take the address of variable `v` by the syntax `&v`. Any scalar (int, float, pointer) local variable whose address is never taken (a property easily to determine statically) is called *nonaddressable*. All other variables—globals, aggregates (`struct`, `union`, arrays), and those whose address is taken—are *addressable*. Addressable variables are stored in memory, while nonaddressable variables are typically kept in machine registers. We pay attention to the difference because, in a program logic, nonaddressable variables can often be reasoned about by substitution, instead of the heavyweight mechanism of separation.

CompCert's front end unfolds C's looping constructs (for, while, do-while) into the loop construct of Clight (and of Verifiable C). In particular, C's `for` loop is not an instance of its `while` loop, since the continue statement jumps to the *increment* rather than to the loop test. In the command "`loop(`c_i`)` c", c is the loop body and c_i is the increment command. Specifically, "`while (`b`)` c" is defined as

```
loop(;) {if (b) /*skip*/; else break; c}
```

and "`for (`c_0`; `b`; `c_i`)` c" is defined as

```
c0; loop(ci) {if (b) /*skip*/; else break; c}
```

2.1 Verifiable C programs

Almost all C light constructs can be verified by Verifiable C, with four limitations:

A. Verifiable C does not support the `goto` statement.
B. Only primary r-value expressions and primary l-value expressions are allowed. A primary r-value expression does not contain any memory dereferences or function calls. A primary l-value expression refers to an address in memory and the computation of the address does not involve any memory dereferences or function calls. For example, if `x` and `y` are nonaddressable variables of integer type and `a` is a variable of integer array type, then `x+y`, `x*x`, `x<=y`, `a+x`, `&(a[x + 1])` are primary r-value expressions; `a[0]`, `a[x]`, `a[x+1]`, `*(a+x)` are primary l-value expressions; and `a[x]+a[y]`, `a[x]*2`, `a[a[x]]` are not primary expressions. When the source-language command has nonprimary expressions, the clightgen tool automatically factors the command (inserting extra assignments to temporary variables), so the user does Verifiable C proofs on abstract-syntax trees that have only primary expressions.
C. Only the following four kinds of assignment commands are allowed:
    1. Set command: the left side is a nonaddressable variable and the right side is a primary r-value expression
    2. Load command: the left side is a nonaddressable variable and the right side is a primary l-value expression
    3. Store command: the left side is a primary l-value expression and right side is a primary r-value expression
    4. Function call: the left side (if present) is a nonaddressable variable and the right side is a function call (perhaps wrapped in a cast).

Again, where the source program contains more general assignments, clightgen factors them into commands of this form before the user does a Verifiable C proof.

D. Test expressions of `if` commands, expressions in `return` commands, and arguments in function calls must be primary r-value expressions. The clightgen tool handles this as well.

Verifiable C imposes these limitations for simplicity of the logic (see *Remark* at end of §2.3). And these limitations do not decrease the expressivity of C language: any nonprimary expression can be split into multiple assignment commands with the help of auxiliary nonaddressable variables. Program transformations can eliminate `goto` commands, although less conveniently.

2.2 Contextual Hoare logic

Verifiable C uses a contextual Hoare triple: $\Delta \vdash \{P\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}$ to describe the behavior of a C command c. The context Δ contains the types of all C variables (accessible from the current function) and the specifications of all C functions (that might be directly called from the current function). Four different post-conditions are involved in one triple because C (or C light) semantics allows to exit the execution of c in four different ways: exit normally, exit by a `break` command, exit by a `continue` command and exit by a `return` command.

The triple means that for any initial state s, if $s \vDash P$, then running c under context Δ from s is safe (i.e., it will not cause a run time error or undefined behavior) and for any possible ending state t of this execution, $t \vDash Q$, $t \vDash Q_{\text{brk}}$, $t \vDash Q_{\text{con}}$ or $t \vDash Q_{\text{ret}}$ if the execution ends normally, by `break`, by `continue` or by `return` respectively.

Verifiable C's program logic is very similar to normal Hoare logic. The structural Hoare rules are as follows.

$$\text{SEMAX-SEQ}\ \frac{\Delta \vdash \{P\}\ c_1\ \{Q, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\} \quad \Delta \vdash \{Q\}\ c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}{\Delta \vdash \{P\}\ c_1; c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}$$

$$\text{SEMAX-PRE}\ \frac{P \vdash P' \quad \Delta \vdash \{P'\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}{\Delta \vdash \{P\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

$$\text{SEMAX-POST}\ \frac{Q' \vdash Q \quad Q'_{\text{brk}} \vdash Q_{\text{brk}} \quad Q'_{\text{con}} \vdash Q_{\text{con}} \quad Q'_{\text{ret}} \vdash Q_{\text{ret}} \quad \Delta \vdash \{P\}\ c\ \{Q', Q'_{\text{brk}}, Q'_{\text{con}}, Q'_{\text{ret}}\}}{\Delta \vdash \{P\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

$$\text{SEMAX-IF}\ \frac{\Delta \vdash \{P \wedge [\![b]\!] = \text{true}\}\ c_1\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\} \quad \Delta \vdash \{P \wedge [\![b]\!] = \text{false}\}\ c_2\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}{\Delta \vdash \{P \wedge \text{tc\_expr}(\Delta, b)\}\ \texttt{if}\ (b)\ c_1\ \texttt{else}\ c_2\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

$$\text{SEMAX-LOOP}\ \frac{\Delta \vdash \{P\}\ c\ \{P', Q, P', Q_{\text{ret}}\} \quad \Delta \vdash \{P'\}\ c_i\ \{P, \bot, \bot, Q_{\text{ret}}\}}{\Delta \vdash \{P\}\ \texttt{loop}\ (c_i)\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

We use $[\![e]\!]$ to represent the value of expression e in the current local-variable state; since e is a primary r-value then it is independent of memory. However, e might still fail to evaluate if it refers to an uninitialized local variable or (e.g.,) divides by zero. The function $\mathsf{tc\_expr}(\Delta, e)$ "type-checks" the expression e: if $\mathsf{tc\_expr}(\Delta, e)$ then e is a primary r-value expression and the evaluation does not involve undefined behavior. Furthermore, tc_expr is *efficient*: in most cases, what the assertion $\mathsf{tc\_expr}(\Delta, e)$ computes to is simply true.

Similarly, when e is a primary l-value expression, we use $[\![\&e]\!]$ to represent the address it evaluates to, and $\mathsf{tc\_lvalue}(\Delta, e)$ typechecks $\&e$.

2.3 Separation logic

Separation Hoare logic is an extension of Hoare logic designed for programming languages with explicit memory manipulation. To support convenient reasoning about pointer (and array-slice) manipulation in C, Verifiable C is a separation logic.

Separation logic adds separating conjunction $*$ to the assertion language. Suppose program states can be represented as pairs of stack (a function from local variables to values) and heap (also called memory, a partial function from addresses to values)[1] then the semantics of separating conjunction is defined as follows:

$$(s,h) \vDash P * Q \text{ iff there exist } h_1 \text{ and } h_2 \text{ s.t.}$$
$$h_1 \oplus h_2 = h, (s,h_1) \vDash P \text{ and } (s,h_2) \vDash Q$$

where $\oplus$ represents the disjoint union of heaps. Intuitively, $P * Q$ is satisfied on a piece of memory if it can be split into two parts, one of which satisfies P and the other satisfies Q.

Separation logic has two advantages over normal Hoare logic:

First, separation logic allows for a concise representation of non-aliasing. For example, $P_1 * P_2 * ... * P_n$ claims that these n assertions are satisfied on n disjoint pieces of memory. In propositional logic, the $n(n-1)/2$ antialiasing claims might require a quadratically long formula.

Second, separation Hoare logic has an extra Hoare rule, the frame rule, which enables one to prove a triple locally and use it globally.

$$\text{SEMAX-FRAME}\ \frac{\Delta \vdash \{P\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\} \qquad \text{closed\_wrt\_modvars}(c, F)}{\Delta \vdash \{F * P\}\ c\ \{F * Q, F * Q_{\text{brk}}, F * Q_{\text{con}}, F * Q_{\text{ret}}\}}$$

The following are Hoare rules for atomic commands in Verifiable C:

$$\text{SEMAX-SET}\ \frac{P \vdash (\mathsf{tc\_expr}(\Delta, e)\ \wedge\ [\![e]\!] = v)}{\Delta \vdash \{P\}\ \ \mathsf{x}{=}e\ \ \{\exists x'.\, [\![\mathsf{x}]\!] = v \wedge P[x'/\mathsf{x}], \bot, \bot, \bot\}}$$

$$\text{SEMAX-LOAD}\ \frac{P \vdash (\mathsf{tc\_lvalue}(\Delta, e)\ \wedge\ [\![\&e]\!] = p\ \wedge\ p \mapsto v\ *\ \top)}{\Delta \vdash \{P\}\ \ \mathsf{x}{=}e\ \ \{\exists x'.\, [\![\mathsf{x}]\!] = v \wedge P[x'/\mathsf{x}], \bot, \bot, \bot\}}$$

[1] The "heap" in Verifiable C is actually a step indexed model of CompCert's memories, following Hobor et al. [17].

$$\text{SEMAX-STORE}\ \frac{P \vdash (\mathsf{tc\_lvalue}(\Delta, e_1)\ \wedge\ [\![\&e_1]\!] = p\ \wedge\ \mathsf{tc\_expr}(\Delta, e_2)\ \wedge\ [\![e_2]\!] = v)}{\Delta \vdash \{P * p \mapsto \_\}\ \ e_1 \texttt{=} e_2\ \ \{P * p \mapsto v, \bot, \bot, \bot\}}$$

Here, p and v range over terms that are independent of program states; $p \mapsto v$ is a separation logic predicate which describes a heap consisting of only one single address p, at which the value v is stored. Remember that "$*$" represents separation of memory access, so $p \mapsto v * q \mapsto u$ implies that p and q are different addresses. These rules can be applied without needing to separately apply the frame rule, because of the $*\top$ in SEMAX-LOAD and the $P*$ in SEMAX-STORE.

Remark. In §2.1, we only allow 4 simple kind of assignment commands and only allow primary r-value expression outside assignment commands. Our separation Hoare logic rules benefit from this setting: evaluations like $[\![e]\!]$ and $[\![\&e]\!]$ are heap independent and only one memory block (described by $p \mapsto v$) is critical for every load/store command.

2.4 Shallowly embedded logic

Verifiable C is a program logic shallowly embedded into Coq. In other words, every assertion P is a predicate on stack-heap pairs.[2] The satisfaction relation is defined straightforwardly:

$$(s, h) \vDash P \quad ::= \quad P(s, h)$$

Shallow embedding enables Verifiable C to represent quantifiers. In the assertion

$$\forall x : A.\ P(x)$$

P has type $(A \to \text{assertion})$ and the universal quantifier in the object language can be directly defined as follows:

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

Notation Distinction: In order to distinguish connectives in object language and metalanguage, we will always use symbols ($\forall$, $\wedge$, $*$) in object language (separation logic assertions) in this paper, while English words ("and", "or", "for all") will represent metalanguage (Coq) connectives.

It is easy to do natural deduction with quantifiers in a shallowly embedded logic. For example, the following is the generalization rule for the universal quantifier formalized in Coq.

```
Lemma allp_right: forall A P Q,  (forall a: A, P ⊢ (Q a))  →  P ⊢ ∀ a: A, Q a.
```

In a Coq proof scenario as below, (apply allp_right; intro a), a tactic in Coq, will turn the proof goal on the left into the proof goal on the right.

[2] Readers can understand the type of P as stack → heap → Prop. But actually, this predicate must be monotonic w.r.t. the step indexing, i.e. we define it Coq as a dependent pair of a predicate and a proof of monotonicity [3, Part V].

$$\frac{}{P \vdash \forall\ a\colon A, Q\ a.} \qquad \frac{a\colon A}{P \vdash Q\ a.}$$

It is particularly convenient to use variables of the metalogic to represent variables of the object language. In comparison, in a fully deeply embedded logic one might have to write the generalization rule for universal quantifiers as,

$$\frac{\Gamma, a : A \mid P \vdash Q[a/x] \qquad v \notin \Gamma}{\Gamma \mid P \vdash \forall x : A.\ Q}$$

in which a and all other variables in Coq's proof context (Γ) are symbols of object language. But the variable a in allp_right is a Coq variable, i.e. a metalogic variable and object logic constant.

Using this design, the only object logic variables in Verifiable C are C program variables. All other variables in paper proofs are formalized in Coq as metalogic variables. For example, our SEMAX_SET rule is formalized as follows:

```
Theorem semax_set:
  forall Δ (P: assertion) x e (v: val),
    P ⊢ tc_expr(e) ∧ ⟦e⟧ = v   →   Δ ⊢ {P} x := e {EX x': val, ⟦x⟧ = v ∧ P[x'/x]}.
```

Here, the term v is formalized as a Coq variable since it represents values independent of program states. The term P[x'/x] is semantic substitution, i.e.

$$P[x'/x] = \lambda(s, h).P(s[x'/x], h)$$

2.5 Unlifted logic: stack-independent heap predicates

In the separation logic of Verifiable C, assertions independent of stacks are widely used in verification. For example, if p and v are values, then $p \mapsto v$ is a predicate independent of stacks. In comparison, $\llbracket \mathsf{x} \rrbracket \mapsto 0$ is not independent of stacks, since it relies on the current value of the local variable x.

In a shallowly embedded logic, these assertions independent of stacks are just predicates over heaps. These predicates also form a separation logic, i.e.

$$h \vDash P * Q \text{ iff there exist } h_1 \text{ and } h_2 \text{ s.t. } h_1 \oplus h_2 = h, h_1 \vDash P \ and\ h_2 \vDash Q$$

We call this separation logic unlifted and we call the previous one lifted. The lifted separation logic and unlifted separation logic have the following connection.

$$\begin{aligned}
P \wedge Q &= \lambda s : \mathsf{stack}.\ P(s) \wedge Q(s) \\
P * Q &= \lambda s : \mathsf{stack}.\ P(s) * Q(s) \\
\exists x : A.\ P(x) &= \lambda s : \mathsf{stack}.\ \exists x : A.\ P(x, s) \\
\forall x : A.\ P(x) &= \lambda s : \mathsf{stack}.\ \forall x : A.\ P(x, s)
\end{aligned}$$

Here, all connectives on the left side are logical connectives of the lifted logic while the connectives on the right side belong to the unlifted logic. The situation with other connectives is similar; we omit them here.

It is particularly useful to describe data structures using heap-only (unlifted) predicates. The list or tree you build now, in this function with these local-variable values, will still be the same list or tree in some other function with a different stack context. Our canonical forms in Section 4 facilitate this decomposition of assertions into a stack-relevant part and a heap-relevant part.

2.6 Function specifications and the function call rule

Because of the shallow embedding, function specifications in Verifiable C are not only pre/postconditions but parameterized pre/postconditions. For example, suppose `swapint` has the following signature in C:

```
void swapint(int * x; int * y);
```

when we informally write[3]

$$\texttt{swapint(x, y)} : \{[\![\mathsf{x}]\!] \mapsto a * [\![\mathsf{y}]\!] \mapsto b\}\{[\![\mathsf{x}]\!] \mapsto b * [\![\mathsf{y}]\!] \mapsto a\}$$

we mean that this specification is valid no matter what values a and b are instantiated (the same pair (a, b) should be used to instantiate the precondition and the postcondition). In Verifiable C, its pre/postconditions are actually

$$\lambda(a, b).\ [\![\mathsf{x}]\!] \mapsto a * [\![\mathsf{y}]\!] \mapsto b \quad \text{and} \quad \lambda(a, b).\ [\![\mathsf{x}]\!] \mapsto b * [\![\mathsf{y}]\!] \mapsto a$$

whose types are "val $\times$ val $\to$ assertion".

To indicate that the specification is parameterized, we will write them in the following way:

$$\texttt{swapint(x, y)} : \Pi(a, b).\ \{[\![\mathsf{x}]\!] \mapsto a * [\![\mathsf{y}]\!] \mapsto b\}\{[\![\mathsf{x}]\!] \mapsto b * [\![\mathsf{y}]\!] \mapsto a\}$$

In VST-Floyd, in Coq notation rather than math notation, we would write this as[4],

DECLARE `swapint`
WITH a: val, b: val
PRE [x OF `int` , y OF `int`] $[\![\mathsf{x}]\!] \mapsto a * [\![\mathsf{y}]\!] \mapsto b$
POST [`void`] $[\![\mathsf{x}]\!] \mapsto b * [\![\mathsf{y}]\!] \mapsto a$

[3] This specification of `swapint` is not strong enough, because it does not say whether the values of $[\![\mathsf{x}]\!]$ and $[\![\mathsf{y}]\!]$ change or not after running the function. In actual verification, we will use:

$$\texttt{swapint(x, y)} : \{[\![\mathsf{x}]\!] = p \wedge [\![\mathsf{y}]\!] = q \wedge p \mapsto a * q \mapsto b\}\{p \mapsto b * q \mapsto a\}$$

[4] In actual Coq code, CLight uses identifiers to represent C variable names and C function names. So, when we write `swapint`, the real Coq code is "_swapint" which is an identifier, i.e. a positive number, in Coq. Similarly, the real Coq code for `int` is "tint" whose type is Clight.type, a Coq inductive type representing the syntax tree of C types.

Here, [x OF `int` , y OF `int`] represent the argument list of the function and [`void`] is the return type.

Generally, a specification can be parameterized by variables of any type, so a Verifiable C's function specification has type

$$\sum_{A \in \mathsf{Type}} (A \to \mathsf{assertion}) \times (A \to \mathsf{assertion})$$

where, in our swap example, $A = \mathsf{val} \times \mathsf{val}$.

Verifiable C has a rule for each kind of C command; here is the rule for function calls.

$$\textsc{semax-call}\ \frac{\begin{array}{c} R * P(a)[\overrightarrow{v}/\overrightarrow{\mathsf{y}}] \vdash \mathsf{tc\_lvalue}(\Delta, \overrightarrow{e}) \ \wedge\ [\![\overrightarrow{e}]\!] = \overrightarrow{v} \\ f(\overrightarrow{\mathsf{y}}) : \Pi a : A. \{P(a)\}\{Q(a)\} \in \Delta \end{array}}{\Delta \vdash \{R * P(a)[\overrightarrow{v}/\overrightarrow{\mathsf{y}}]\}\ \ \mathsf{x} = f(\overrightarrow{e})\ \ \{R * Q(a)[\mathsf{x}/\mathsf{ret\_val}], \bot, \bot, \bot\}}$$

2.7 From C command verification to program verification

In Verifiable C, a function specification is verified if the corresponding Hoare triple of its implementation is proved. Formally,

$$\begin{array}{l} \Delta \vdash f(\overrightarrow{\mathsf{x}}) : \Pi a : A.\ \{P(a)\}\{Q(a)\} ::= \\ \quad \text{for any } a \in A, \\ \qquad \Delta \vdash \{P(a) * \mathsf{LocalVar}(f)\} \text{ body of } f\ \{\bot, \bot, \bot, Q(a) * \mathsf{LocalVar}(f)\} \end{array}$$

Here, $\mathsf{LocalVar}(f)$ is the spatial assertion for f's addressable local variables. For example, if this function f has one local addressable variable p of type `int` (suppose it is addressable because its address is requested in the function body), then $\mathsf{LocalVar}(f)$ is $\exists v.[\![\&\mathsf{p}]\!] \mapsto v$.

The correctness of a C program is composed from the correctness of all its functions. Specifically, the main process of verifying a C program with n functions f_1, f_2, ..., f_n is as follows.

1. Use parameterized pre/postconditions to specify all functions. Assume these specifications are:
$$f_i(\overrightarrow{\mathsf{x}_i}) : \Pi a : A_i.\ \{P_i(a)\}\{Q_i(a)\}$$
Here, $\overrightarrow{\mathsf{x}_i}$ is the list of the parameter names of function f_i.
2. Verify every function, i.e. prove $\Delta_i \vdash f_i(\overrightarrow{\mathsf{x}_i}) : \Pi a : A_i.\ \{P_i(a)\}\{Q_i(a)\}$ for all i. Here, each Δ_i includes *all* the function specifications of all the f_j, plus type specifications for the local variables of f_i and type specifications for all global variables.

Since function specifications appear both as assumptions (in contexts) and as conclusion in the steps above, we have to worry about the wellfoundedness of the self-reference in this verification process. Verifiable C solves this problem by using step-indexing [4]; details are hidden from users (the step indexes are in the model by which the Hoare logic is proved sound, not in the Hoare logic itself). Verifiable C allows recursive and even mutually recursive C functions.

3 Concise Separation Logic Predicates For C Aggregate Types

In Reynolds's original version of separation logic, he [29] used abbreviations like $p \mapsto a, b$ for conciseness, to mean $p \mapsto a * p + 1 \mapsto b$. To improve the usability of Verifiable C beyond atomic maps-to predicates $p \mapsto v$, VST-Floyd provides separation logic predicates for structured data. Specifically, $p \underset{t}{\longmapsto} v$ or (data_at t v p) in Coq[5], says that a datum v of C type t is stored at address p. Here, t is the AST (syntactic description) of any C-language type expression, including integers, floats, pointers, `struct`, `union` and array.

The assertion $p \underset{t}{\longmapsto} v$ is dependently typed: the type of v depends on the value of t. When t is a struct, v is a tuple. For example

```
struct IntPair { int fst; int snd; };
```

$$p \underset{\texttt{IntPair}}{\longmapsto} a, b \;\vdash\; p \mapsto a \;*\; p + 4 \mapsto b$$

An integer occupies 4 bytes, so the second field is stored at $p + 4$. (The reader might find $p + 4$ insufficiently abstract; later we describe an addressing abstraction, field_address.) The entailment shown is not an equation! The left side is stronger than the right side. The left side, our new predicate data_at, also enforces alignment and end-of-memory constraints for C structured data. In this example, alignment enforces that address p is a multiple of 4 (sizeof(int)) and $p+8$ is not beyond the end of memory.

Similarly, when t is a `union`, v's type is a sum type. When t is an array, v is a list. For example,

$$p \underset{\texttt{int[3]}}{\longmapsto} [a; b; c] \vdash p \mapsto a \;*\; p + 4 \mapsto b \;*\; p + 8 \mapsto c$$

It's no coincidence that the length of $[a; b; c]$ matches the declared array size 3; this is enforced by data_at, else the predicate is equivalent to false.

Struct, union and array can be nested. For example, the following predicate describes an array of `struct` type.

$$p \underset{\texttt{IntPair[3]}}{\longmapsto} [(a_1, a_2); (b_1, b_2); (c_1, c_2)]$$

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

[5] Our $\mapsto$ and data_at predicates take another argument that we omit in this article: a permission-share indicating read-only, read-write, or various other levels of access to the data.

3.1 Proof theory of data_at and field_at.

Besides $p \underset{t}{\longmapsto} v$, VST-Floyd provides another predicate $p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} v$, or (field_at t $\overrightarrow{f}$ v p) in Coq. It says, starting from address p, follow the path $\overrightarrow{f}$ of field-selection/array-indexing to arrive at a memory datum v of type t. For example,

$$p \xmapsto[\texttt{IntPair}]{} a, b \;=\; p \xmapsto[\texttt{IntPair}]{\texttt{[fst]}} a \;*\; p \xmapsto[\texttt{IntPair}]{\texttt{[snd]}} b$$

$$p \xmapsto[\texttt{IntPair[3]}]{\texttt{[1]}} b_1, b_2 \;=\; p \xmapsto[\texttt{IntPair[3]}]{\texttt{[1;fst]}} b_1 \;*\; p \xmapsto[\texttt{IntPair[3]}]{\texttt{[1;snd]}} b_2$$

The *nested field* $\overrightarrow{f}$ is a path of general fields (a `struct` field, a `union` field or an array subscript).

The proof theory of data_at and field_at has unfolding rules and a reroot lemma, shown in Figure 3. The unfolding rules composed transform data_at into separat-

$$p \underset{t}{\longmapsto} v \;=\; p \underset{t}{\overset{[]}{\longmapsto}} v \tag{1}$$

$$p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} v \;=\; p \triangleright \overrightarrow{f} \mapsto v \quad \text{if } t.\overrightarrow{f} \text{ is an elementary type} \tag{2}$$

$$p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} v \;=\; \bigstar_{f \in t.\overrightarrow{f}} \left(p \underset{t}{\overset{\overrightarrow{f}\,f}{\longmapsto}} v.f * \text{Space}(\overrightarrow{f}\,f, p) \right) \quad \text{if } t.\overrightarrow{f} \text{ is a nonempty struct} \tag{3}$$

$$p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} \{f : v\} \;=\; p \underset{t}{\overset{\overrightarrow{f}\,f}{\longmapsto}} v * \text{Space}(\overrightarrow{f}\,f, p) \quad \text{if } t.\overrightarrow{f} \text{ is a union} \tag{4}$$

$$p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} v \;=\; \bigstar_{0 \le i < n} p \underset{t}{\overset{\overrightarrow{f}\,i}{\longmapsto}} v_i \quad \text{if } t.\overrightarrow{f} \text{ is an array of positive length } n \tag{5}$$

$$p \underset{t}{\overset{\overrightarrow{f}}{\longmapsto}} v \;=\; p \triangleright \overrightarrow{f} \xmapsto[t.\overrightarrow{f}]{} v \tag{6}$$

Fig. 3: Unfolding rules and reroot lemma

ing conjunctions of ordinary maps-to predicates. In Figure 3, equation (1) says that data_at is field_at with empty field path. Equation (2) says that field_at on elementary types (integers, floating point numbers, pointers) is equivalent to an ordinary maps-to predicate with an offset. Equations (3), (4), and (5) are single-layer unfolding rules. Here, we use $t.\overrightarrow{f}$ to represent the type of field $\overrightarrow{f}$ in t and we use the Space predicate to implement C's alignment rules for `struct` fields and `union` fields.

Equation (6) is the reroot equation: its left side is a predicate on an internal node of a "tree" and the right side treats the internal node as a root. For example,

$$p \xmapsto[\texttt{IntPair}]{\texttt{[fst]}} a \;=\; p \triangleright [\texttt{fst}] \xmapsto[\texttt{int}]{} a$$

We use $p \triangleright \overrightarrow{f}$ to represent an address with an offset (in Coq, written (field_address t $\overrightarrow{f}$ p), where t is the type of p). Specifically, we define $p \triangleright \overrightarrow{f}$ as

the starting address of nested field $\overrightarrow{f}$ when the base address of the entire aggregate is at p. If

1. p is a legal starting address for type t, i.e. (1a) there is enough space in heap from p to store a data of type t and (1b) p is a multiple of the alignment of type t
2. $\overrightarrow{f}$ is a legal nested field of t, i.e. (2a) struct or union fields in the path are fields in structure definition and (2b) array subscripts are in range

then $p \triangleright \overrightarrow{f}$ is equivalent to $p + \delta(\overrightarrow{f})$, where $\delta(\overrightarrow{f})$ represents the offset of $\overrightarrow{f}$. In VST-Floyd, we use field_compatible to represent conditions (1) and (2) above. Thus, the meaning of $p \triangleright \overrightarrow{f}$ is,

$$p \triangleright \overrightarrow{f} = p + \delta(\overrightarrow{f}) \quad \text{if field\_compatible}(t, \overrightarrow{f}, p)$$
$$p \triangleright \overrightarrow{f} = \text{Vundef} \quad \text{otherwise}$$

3.2 Coq implementation of data_at and field_at.

In Coq, the predicate $p \underset{t}{\longmapsto} v$ (or data_at) is typed as follows.

```
reptype : Clight.type → Type
data_at : forall t : Clight.type, reptype t → val → pred heap
```

The function reptype means "representation type"; it translates from a syntactic description of a C type to a Coq Type. A C array is represented as a Coq list, a `struct` is represented as a tuple, and a `union` is represented as a sum. By (pred heap) in Coq , we mean predicates over heaps.

Using this concept, $p \overset{\overrightarrow{f}}{\underset{t}{\longmapsto}} v$ (or field_at) is typed as follows.

```
nested_field_type: Clight.type → list gfield → Clight.type
field_at : forall (t : Clight.type) (path: list gfield),
                  reptype (nested_field_type t path) → val → pred heap
```

Here, gfield means "general field" and a general field can be a `struct` field, a `union` field or an array subscript. Thus a list of gfields represents a path from a root type to a field type. The function nested_field_type computes that field type from the given root type and the given path of general fields. We write $t.\overrightarrow{f}$ as an abbreviation of $(\text{nested\_field\_type}\ t\ \overrightarrow{f})$.

We first define an auxiliary predicate $p \underset{t}{\dashrightarrow} v$, or (data_at_rec t v p), as a recursive function in Coq. Then we define field_at as an instance of data_at_rec and

define data_at as field_at with empty field path. Specifically:

$$p \vdash\!\!\xrightarrow[t]{}\ v \ ::=\ p \mapsto v \quad \text{if } t \text{ is integer, float, a pointer}$$

$$p \vdash\!\!\xrightarrow[\texttt{struct } t]{}\ v \ ::=\ \bigstar_{f \in t} \left(p + \delta(f) \vdash\!\!\xrightarrow[t.f]{}\ v.f \ *\ \mathsf{Space}([f], p) \right)$$

$$p \vdash\!\!\xrightarrow[\texttt{union } t]{}\ \{f : v\} \ ::=\ p \vdash\!\!\xrightarrow[t.f]{}\ v \ *\ \mathsf{Space}([f], p)$$

$$p \vdash\!\!\xrightarrow[t[n]]{}\ v \ ::=\ \bigstar_{0 \le i < n} \left(p + i \cdot \mathsf{sizeof}(t) \vdash\!\!\xrightarrow[t]{}\ v_i \right)$$

$$p \vdash\!\!\xrightarrow[t]{\overrightarrow{f}} v \ ::=\ p + \delta(\overrightarrow{f}) \vdash\!\!\xrightarrow[t.\overrightarrow{f}]{}\ v \ \wedge\ \mathsf{field\_compatible}_t(\overrightarrow{f}, p)$$

$$p \longmapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow\ v \ ::=\ p \vdash\!\!\xrightarrow[t]{[]} v$$

Here, $v.f$ represents the element indexed by f when v is a tuple. When v is a list, v_i represents the element indexed by i.

We *could* let data_at be defined as a conjunction of data_at_rec and field_compatible directly and let the reroot equation be the definition of field_at. However, treating data_at as field_at with empty path enables us to handle both constructions uniformly in our tactics.

Reptype and data_at_rec are implemented as Coq functions recursive on C light type[6]. Nested_field_type and nested_field_o set ($\delta(f)$) are implemented as Coq functions recursive on the path.

4 Canonical Form

In this section, we introduce the canonical form of assertions which plays an important role in strongest postcondition generation and our implementation of forward verification tactics.

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

PROP: pure propositions that are independent of stack and heaps
LOCAL: values of nonaddressable variables and addresses of addressable variables
SEP: spatial separation-logic predicates that are independent of stacks

[6] In CompCert 2.4 and earlier versions, the C light type definition is a Coq inductive type. However, from CompCert 2.5, `struct` and `union` types are represented by name instead of by structure. Specifically, every C light program is associated with a composite_env. A composite_env is a dictionary mapping every `struct`/`union` name to a list of all its fields. The meaning of a `struct` or a `union` needs to be interpreted by looking it up in the dictionary. From then on, reptype and data_at_rec are no longer Coq functions recursive on Coq inductive structure. The CompCert developers accepted our suggestion that every type should be tagged with a rank, which is a natural number. The ranking system ensures that the rank of a struct type is the max rank of its fields plus one; the rank of a union type is the max rank of its fields plus one; the rank of an array type is the rank of its element type plus one. The rank of elementary types (including pointers) is zero. Our current definition of reptype and data_at_rec are recursive functions on this rank.

Formally, a canonical assertion has the form

$$\mathsf{PROP}(P_1; P_2; ..)\mathsf{LOCAL}(Q_1; Q_2; ..)\mathsf{SEP}(R_1; R_2; ..)$$

The P_i have type Prop in Coq. The Q_i are fully syntactic (deeply embedded), and have denotations in type stack→ Prop; and the R_i are predicates over heaps (i.e. assertions in the unlifted separation logic).[7] Every conjunct in the LOCAL part is one of the following:

temp x v, meaning that x is a nonaddressable variable and $[\![\mathsf{x}]\!] = v$
var x v, meaning that x is a local addressable variable and $[\![\&\mathsf{x}]\!] = v$
gvar x v, meaning x is a global variable and $[\![\&\mathsf{x}]\!] = v$.

For example, this assertion in canonical form

$$\mathsf{PROP}(a \geq 0; b \geq 0)\ \mathsf{LOCAL}(\mathsf{temp\ x}\ p)\ \mathsf{SEP}(p \mapsto a; (p+4) \mapsto b)$$

represents $a \geq 0\ \wedge\ b \geq 0\ \wedge\ [\![\mathsf{x}]\!] = p\ \wedge\ p \mapsto a\ *\ (p+4) \mapsto b$.

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

Other than this restriction, the canonical form is flexible in its PROP and SEP parts. In the rest of this section, we demonstrate some examples of triples (subsection 4.1) and function specifications (subsection 4.2) in canonical form. At the end of this section, we discuss the expressiveness of canonical assertions in subsection 4.3.

4.1 Examples: triples of atomic C commands

Fig. 4 shows a program fragment, to demonstrate Hoare triples of set, load and store commands. The assertions in this decorated program are all written in canonical form, and the postconditions are strongest postconditions.

4.2 Examples: function specifications

Suppose we have a C function with the following signature:

```
void swapIntpair(struct IntPair * x);
```

where IntPair the struct from Section 3. Then the following specification says that the numbers stored in the two fields will be swapped:

$$\texttt{swapIntpair}: \ \Pi\, a\, b\, p.$$
$$\{\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ x}\ p)\mathsf{SEP}(p \xmapsto[\texttt{IntPair}]{} a, b)\}$$
$$\{\mathsf{PROP}()\mathsf{LOCAL}()\mathsf{SEP}(p \xmapsto[\texttt{IntPair}]{} b, a)\}$$

[7] In the Coq development, we use the name environ for what we call "stack" in the paper.

```
int *x; int y, z;
```
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ p;\mathsf{temp\ y}\ a)\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} 0)\}$
```
  z = * x;
```
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ p;\mathsf{temp\ y}\ a;\mathsf{temp\ z}\ b)\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} 0)\}$
```
  y = y + z;
```
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ p;\mathsf{temp\ y}\ (a+b);\mathsf{temp\ z}\ b)\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} 0)\}$
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ p;\mathsf{temp\ y}\ (a+b))\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} 0)\}$
```
  x = x + 1;
```
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ (p+4);\mathsf{temp\ y}\ (a+b))\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} 0)\}$
```
  * x = y;
```
$\{\mathsf{PROP}()\,\mathsf{LOCAL}(\mathsf{temp\ x}\ (p+4);\mathsf{temp\ y}\ (a+b))\,\mathsf{SEP}(p \xmapsto[\mathtt{int}]{} b;\ p+4 \xmapsto[\mathtt{int}]{} a+b)\}$

Fig. 4: Program annotated with Hoare assertions

Sometimes it is useful to wrap an existential quantifier around a canonical-form assertion.

```
void sort(int * bg; int * ed);
```
$\mathtt{sort}:\ \Pi\, l\ p\ n.$
$\{\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ bg}\ p;\mathsf{temp\ ed}\ (p+4n))\mathsf{SEP}(p \xmapsto[\mathtt{int}[n]]{} l)\}$
$\{\exists l'.\mathsf{PROP}(\mathsf{Permutation}(l,l');\mathsf{ordered}(l'))\mathsf{LOCAL}()\mathsf{SEP}(p \xmapsto[\mathtt{int}[n]]{} l')\}$

User-defined predicates *in the unlifted separation logic* can describe data structures in memory. For example, the following C `struct` is a C structure for linked lists of integers.

```
struct IntList { int num; struct IntList * link };
```

Users can define a separation logic predicate for this data structure:

$$\mathrm{list}(p,l) ::= \begin{cases} \text{if } l = \mathsf{nil},\ \ p = \mathsf{null} \quad \wedge\ \ \mathsf{emp} \\ \text{if } l = hd :: tl, \quad \exists q.\ \ p \xmapsto[\mathtt{IntList}]{} hd, q \ \ * \ \ \mathrm{list}(q, tl) \end{cases}$$

Such definitions can be easily formalized in Coq:

```
Fixpoint list (p: val) (l: list int): pred heap :=
  match l with
  | nil ⇒ (fun _ ⇒ p = null) ∧ emp
  | hd :: tl ⇒ ∃ q. data_at IntPair (hd, q) p * list q tl
  end.
```

A C function to reverse linked lists can be specified using this predicate:

```
struct IntList * reverse(struct IntList * hd);
```
$\mathtt{reverse}:\ \Pi\, l\ p.$
$\{\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ hd}\ p)\mathsf{SEP}(\mathrm{list}(p,l))\}$
$\{\exists p'.\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ ret\_temp}\ p')\mathsf{SEP}(\mathrm{list}(p', rev(l)))\}$

where rev denotes list-reversal function from Coq's standard library

4.3 Discussion: expressiveness

In the examples that we presented above, we have seen the convenience achieved by limiting the use of C variables in canonical assertions. Specifically, the postcondition of a set command or a load command only modifies or adds one conjunct in the LOCAL clauses. With a store command, the postcondition's SEP clause differs only in one term from the precondition's. It is natural to ask whether this setting restricts the expressiveness of the assertion language. We answer this question here.

Observation one: any assertion can be decomposed into existentially quantified canonical form. Let P be any assertion about (nonaddressable) local variables $\mathsf{x}_1, \mathsf{x}_2, \cdots, \mathsf{x}_n$. Then P can be decomposed into,

$$\exists x_1 x_2 \ldots x_n.$$
$$\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp}\ \mathsf{x}_1\ x_1; \ldots \mathsf{temp}\ \mathsf{x}_n\ x_n)\ \mathsf{SEP}(P[x_1/[\![\mathsf{x}_1]\!], \ldots, x_n/[\![\mathsf{x}_n]\!]])$$

Addressable local and global variables can be substituted using lvar, gvar and sgvar.

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

$$\text{EXTRACT-EXISTS}\frac{\forall x : A.\ (\Delta \vdash \{P(x)\}\ c\ \{Q\})}{\Delta \vdash \{\exists x : A.\ P(x)\}\ c\ \{Q\}}$$

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

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

5 Sound and Efficient Postcondition Generation

In Fig. 4, we illustrated how atomic commands can be concisely characterized by Hoare triples with canonical assertions. Now we demonstrate how those postconditions in triples can be generated. VST-Floyd produces strongest postconditions, with soundness proofs for the corresponding triples. The property of being *strongest* postconditions is not proved in Coq; it is only a meta-property ensuring that any sound Hoare triple can be proved by Floyd.

Formally speaking, when the command c is a set, load, store, or function call, our tactics in Coq prove the following proposition, instantiating the unification variable ?Post with a strongest postcondition.

$$\Delta \vdash \left\{\mathsf{PROP}()\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ \ c\ \ \{?\mathsf{Post}, \bot, \bot, \bot\}$$

Because an assignment statement or function call does not `break`, `continue`, or `return`, those three postconditions can be false. Only the postcondition for normal termination is nontrivial. We assume that every C variable appears at most once in $\overrightarrow{Q}$.

Otherwise, e.g. if $\overrightarrow{Q}$ were [temp x x_1; temp x x_2], we could remove a conjunct and add $x_1 = x_2$ into the PROP part. During symbolic execution, canonical preconditions have empty PROP clauses because they can all be moved "above the line" into Coq assumptions of the whole triple. Still, PROP part is useful in other assertions like existentially quantified canonical assertions and precondition in parameterized function specifications.

For handling these atomic commands, there are two common issues. One is deriving C expression evaluation. For example, if c is a set command $\mathsf{x} = e$ then in order to generate the strongest postcondition, we need to find a value v such that

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

If c is a store command $e_1 = e_2$ then we need to find values p and v satisfying both of the following criteria

$$\mathsf{PROP}() \ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e_1]\!] = p$$

$$\mathsf{PROP}() \ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![e_2]\!] = v$$

The other common issue is eliminating C variable substitution and its existential quantifier (there may be other existential quantifiers). For example, this rule for load commands was presented in §2.

$$\text{SEMAX\_LOAD}\frac{P \vdash \mathsf{tc\_lvalue}(e) \ \wedge\ [\![\&e]\!] = p \ \wedge\ p \mapsto v \ *\ \top}{\Delta \vdash \{P\}\ \ \mathsf{x}{=}e\ \ \{\exists x'.\, [\![\mathsf{x}]\!] = v \wedge P[x'/\mathsf{x}], \bot, \bot, \bot\}}$$

These quantifiers and variable substitutions make proofs inconvenient, especially interactive proofs. Actually, this is one of the most important drawbacks of forward verification, compared to backward verification. To solve this problem, we must eliminate them in our strongest postconditions.

In §5.1 and §5.2 we present our solutions to these two common issues. Both of our solutions greatly benefit from using canonical form. Then, we will introduce our tactics that generate strongest postconditions of set commands, load commands, store commands, and function calls.

5.1 Computational derivation of expression evaluation

As motivated above, we want to derive the value of a C expression: given a primary r-value expression e and a canonical precondition $\mathsf{PROP}() \ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}$, we need to find a value v such that

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

Similarly, when e is a primary l-value expression, we need to find a v such that

$$\mathsf{PROP}() \ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e]\!] = v$$

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

$$[\![\mathsf{x}]\!] = 1 \wedge [\![\mathsf{y}]\!] = 1 \wedge [\![\mathsf{z}]\!] = 0 \vdash [\![\mathsf{x} + \mathsf{y}]\!] = v$$

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

```
ρ: stack
H: [[x]] ρ = 1
H0: [[y]] ρ = 1
H1: [[z]] ρ = 0
---------------------------
[[x]] ρ + [[y]] ρ = ?v
```

Then, we can look for useful assumptions and rewrite them in the conclusion, i.e. rewrite H, H0. However, searching useful assumptions needs careful tactic programming. Moreover, both proof searching and rewriting are slow in Coq.

We define two mutually recursive functions msubst_eval_expr (for primary r-value expression) and msubst_eval_lvalue (for primary l-value expression) which do symbolic evaluation independent of stack[8]:

msubst_eval_expr: Clight.expr → list localdef → option val
msubst_eval_lvalue: Clight.expr → list localdef → option val

Their definitions are almost the same as $[\![e]\!]$ and $[\![\&e]\!]$. The only difference is that $[\![e]\!]$ and $[\![\&e]\!]$ look up values of nonaddressable variables and addresses of addressable variables in stacks but msubst_eval_expr and msubst_eval_lvalue look them up in LOCAL clauses. We prove the following lemmas and use them to construct expression evaluation directly.

$$\text{MSUBST-EXPR}\frac{\mathsf{msubst\_eval\_expr}\ e\ \overrightarrow{Q} = \mathsf{Some}\ v}{\mathsf{PROP}()\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![e]\!] = v}$$

$$\text{MSUBST-LVALUE}\frac{\mathsf{msubst\_eval\_lvalue}\ e\ \overrightarrow{Q} = \mathsf{Some}\ v}{\mathsf{PROP}()\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e]\!] = v}$$

The value v will then be generated by computation which is much faster than proof search and rewriting in Coq.

5.2 Symbolic variable substitution

If an atomic command modifies a nonaddressable variable (e.g. a set command or a load command), its forward rule (Robert W. Floyd's assignment rule, to distinguish from C. A. R. Hoare's "backward" assignment rule) contains an existential quantifier and variable substitution (see semax_set in §2.4).

[8] In our Coq development, we actually turn the symbolic LOCAL clauses into binary trees first. Then we look up in these trees during symbolic evaluation. We omit the technical details here.

$$\text{Floyd}\frac{}{\{P\}\ x := E\ \{\exists y.\, x = E[y/x] \wedge P[y/x]\}} \qquad \text{Hoare}\frac{}{\{Q[E/x]\}\ x := E\ \{Q\}}$$

Having variable substitutions in assertions is very inconvenient for later proofs. Fortunately, variable substitution on canonical assertions can be reduced away. Specifically, if x is a nonaddressable variable, then:

$$\frac{(\mathsf{temp\ x}\ v) \in \overrightarrow{Q}}{(\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R})[x_0/\mathsf{x}] \vdash \mathsf{PROP}(x_0 = v; \overrightarrow{P})\mathsf{LOCAL}(\mathsf{remove\_temp}(\mathsf{x}, \overrightarrow{Q}))\mathsf{SEP}\ \overrightarrow{R}}$$

$$\frac{(\mathsf{temp\ x\ \_})\ \textit{does not appear in}\ \overrightarrow{Q}}{(\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R})[x_0/\mathsf{x}] \dashv\vdash \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}}$$

Here, remove_temp is a Coq function which deletes conjuncts about a specific nonaddressable from the local clauses (if there exists one). This rule is sound because: (1) Both PROP and SEP parts are independent of stacks; thus variable substitutions have no effect on them. (2) All LOCAL conjuncts other than "temp x" are not affected by substituting x.

Actually, in the first lemma above, the left side and the right are also equivalent, instead of left implies right, if x does not appear multiple times in the LOCAL part. Later in this section, we will use the direction from left to right to prove soundness of Hoare triples while the direction from right to left ensures that the generated postconditions are strongest postconditions. We do not prove this "strongest" property in Coq. Thus, we do not need to prove right-to-left in Coq either.

As an immediate consequence of two lemmas above, existentially quantified old values can be eliminated together with substitution, i.e.

$$\exists x_0.(\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R})[x_0/\mathsf{x}] \vdash \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}(\mathsf{remove\_temp}(\mathsf{x}, \overrightarrow{Q}))\mathsf{SEP}\ \overrightarrow{R}$$

5.3 Strongest postcondition of set commands

Based on previous theorems of C expression evaluation and symbolic variable substitution, we prove the following derived rule:

Lemma SEMAX-SET-CANON: forall $\Delta\ \overrightarrow{Q}\ \overrightarrow{R}$ x $e\ v$,
PROP() LOCAL $\overrightarrow{Q}$ SEP $\overrightarrow{R} \vdash$ tc_expr $\Delta\ e \rightarrow$
msubst_eval_expr $e\ \overrightarrow{Q}$ = Some $v \rightarrow$
$\Delta \vdash$ { PROP() LOCAL $\overrightarrow{Q}$ SEP $\overrightarrow{R}$ }
x = e
{ PROP() LOCAL(temp x v; remove_temp x $\overrightarrow{Q}$) SEP $\overrightarrow{R}$, $\bot$, $\bot$, $\bot$ }

To solve the following proof goal (e is a primary r-value expression):

$$\left\{\mathsf{PROP}()\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ \mathsf{x}\ =\ e\ \{?\mathsf{Post}, \bot, \bot, \bot\}$$

we run "eapply semax_set_canon" in Coq. Two proof goals will be left. One proof goal is the typechecking requirement. Most of times, it can be solved automatically, i.e. the right side is just true. When e is an expression like `y/z`, then tc_expr(Δ, e) computes to the assertion $[\![\mathtt{z}]\!] \neq 0$, which the user must prove manually (if the tactic automation does not solve it automatically).

The other proof goal is:

$$\mathsf{msubst\_eval\_expr}\ e\ \overrightarrow{Q} = \mathsf{Some}\ ?v$$

We use "reflexivity" in Coq to solve this proof goal. Then the unification variable $?v$ will be filled by the computation of left side.

By all these steps above, together with the derived Hoare rule for set commands, we have already generated a (strongest) postcondition of the required set command and a soundness proof of corresponding triple. The generated postcondition is in canonical form. Its PROP part is empty; its SEP part is the same as in the precondition; the LOCAL part replaces the conjunct of "temp x" with the expression evaluation result.

5.4 Strongest postcondition of load and store commands

In the previous section, we focused on heap-independent C assignment commands, but Verifiable C also provides proof rules for assignments that do loads or stores. A *load* is an assignment `x = e` in which `x` is a nonaddressable variable, e is a primary l-value and (therefore) computing the address of e is heap-independent. A *store* is an assignment e_1 `=` e_2 in which e_1 is a primary l-value, e_2 is a primary r-value and (therefore) computing the address of e_1 and the value of e_2 is heap-independent. C-language assignments where e, e_1, or e_2 are nonprimary can be refactored into a series of loads perhaps followed by a store.

5.4.1 Store command examples

We illustrate with an example taken from a case study, verifying the mbedTLS implementation of AES encryption [14]. The program uses a context struct which is passed to all functions and is defined as follows (simplified):

```
struct aes_context {
   int nr; // number of rounds
   int rk[60]; // round keys
};
```

In the following examples, assume that a pointer named `ctx` of type `struct aes_context*` is in scope. We will now consider different ways of initializing (parts of) the round key array of this context struct:

The most straightforward way is to write out in the assignment command the whole access path to the data:

```
for (i = 0; i < 8; i++) {
   (*ctx).rk[i] = ...                                   (a)
}
```

That is, both `.rk` and `[i]` are on the left-hand side of the assignment statement.

In C, one usually would write `ctx`→`rk` instead of `(*ctx).rk`. VST supports both notations and treats them identically. Here we write `(*ctx).rk` to illustrate the *path* `.rk[i]` more clearly. Although `(*ctx)` looks like a memory dereference, when treated as an l-value it is effectively `(&*ctx)`, which is like `(ctx)`; there is no load or store in the computation of the l-value on the left-hand-side of the assignment.

The actual AES implementation optimizes by precomputing the `.rk` part of the path before the loop:

```
int *p = (*ctx).rk;
for (i = 0; i < 8; i++) {
   p[i] = ...
}
```
(b)

As we can see, only part of the access path (namely `[i]`) is visible in the assignment command, while the first part of the access path is above the loop. Another common pattern is to increment a pointer in each loop iteration:

```
int *p = (*ctx).rk;
for (i = 0; i < 8; i++) {
   *p = ...
   p++;
}
```
(c)

Here, the array index is not written explicitly in the source code. But C code can be even more intricate than that: In the next example, adapted from a case study on a garbage collector, we have a memory access with a seemingly negative array index:

```
int *p = (*ctx).rk + 1;
for(i = 0; i < 8; i++) {
   p[-1] = ...
   p++;
}
```
(d)

Given a precondition and a C store command, the goal of strongest postcondition generation is to find the SEP clause affected by the store, and to update the modified substructure of its value with the assigned value. For instance, for the store command in example (b), it should identify the (only) SEP clause for a and update the ith entry of its value ℓ (which is achieved by upd_Znth) with the result v of evaluating the right-hand side of the store command:

$$\begin{array}{l}\{\mathsf{PROP}\,(0 \le i < 8)\\ \ \mathsf{LOCAL}\,(\mathsf{temp}\ \texttt{ctx}\ a;\ \mathsf{temp}\ \texttt{p}\ (a \triangleright [\texttt{rk}]);\ \mathsf{temp}\ \texttt{i}\ i)\\ \ \mathsf{SEP}\,(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} \ell)\}\\ \quad \texttt{p[i]} = e_2\\ \{\mathsf{PROP}\,(0 \le i < 8)\\ \ \mathsf{LOCAL}\,(\mathsf{temp}\ \texttt{ctx}\ a;\ \mathsf{temp}\ \texttt{p}\ (a \triangleright [\texttt{rk}]);\ \mathsf{temp}\ \texttt{i}\ i)\\ \ \mathsf{SEP}\,(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} (\mathsf{upd\_Znth}\ i\ \ell\ v))\}\end{array}$$

For load commands, on the other hand, the SEP clauses remain unchanged, but the LOCAL clause for the modified variable has to be updated. For instance, if we assign `p[i]` to the variable `y`, strongest postcondition generation should figure out

that the value of `p[i]` is defined by the SEP clause for a, and that it can be selected from the value ℓ by taking the ith element ℓ_i, and it should update the LOCAL clause for `y` to this calculated value ℓ_i:

$$\begin{array}{l}\{\mathsf{PROP}\,(0 \le i < 8)\\ \ \mathsf{LOCAL}\,(\mathsf{temp}\ \texttt{ctx}\ a;\ \mathsf{temp}\ \texttt{p}\ (a \triangleright [\texttt{rk}]);\ \mathsf{temp}\ \texttt{i}\ i;\ \mathsf{temp}\ \texttt{y}\ 0)\\ \ \mathsf{SEP}\,(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} \ell)\}\\ \quad \texttt{y = p[i]}\\ \{\mathsf{PROP}\,(0 \le i < 8)\\ \ \mathsf{LOCAL}\,(\mathsf{temp}\ \texttt{ctx}\ a;\ \mathsf{temp}\ \texttt{p}\ (a \triangleright [\texttt{rk}]);\ \mathsf{temp}\ \texttt{i}\ i;\ \mathsf{temp}\ \texttt{y}\ \ell_i))\\ \ \mathsf{SEP}\,(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} \ell)\}\end{array}$$

5.4.2 General load and store rules defined in terms of field_at

The foundational separation logic of Verifiable C provides two basic rules, SEMAX-LOAD and SEMAX-STORE (§2.3), to reason about memory loads and stores. However, their pre- and postconditions are not in canonical form, and they are defined in terms of the mapsto operator, which talks only about primitive values, but not about (possibly nested) structs and arrays like field_at and data_at.

One way to solve this would be to unfold the field_at or data_at into many mapsto assertions, apply SEMAX-LOAD or SEMAX-STORE, respectively, and then fold them back into field_at or data_at. One could automate these steps with tactics, but this would be very slow to run and generate huge proof terms, which take a long time to typecheck during **Qed**.

So instead, we design (and prove) higher-level versions of the SEMAX-LOAD and SEMAX-STORE rules defined in terms of field_at, and using canonical assertions; and we program tactics to infer all parameters needed to apply them; so that we *efficiently* obtain strongest postconditions for load and store commands together with a proof for the claim.

Consider the following load and store rules:[9]

$$\text{LOAD-1}\ \frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e]\!] = q\\ q = a \triangleright \overrightarrow{f} \qquad \overrightarrow{f} = \overrightarrow{f_0}{+\!+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v')\\ (\text{the component in } v' \text{ denoted by } \overrightarrow{f_1}) = v\\ (\overrightarrow{Q} \text{ with the value for } \texttt{x} \text{ updated to } v) = \overrightarrow{Q'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ \texttt{x = } e\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q'}\ \mathsf{SEP}\ \overrightarrow{R}\right\}}$$

[9] All the load and store rules of this section also need typechecking side conditions, i.e., tc_expr(Δ, e) for each involved C expression e. We omit them for brevity.

$$\frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e_1]\!] = q \wedge [\![e_2]\!] = v \\ q = a \triangleright \overrightarrow{f} \qquad \overrightarrow{f} = \overrightarrow{f_0}{+\!+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v_{old}) \\ (v_{\mathrm{old}} \text{ with the substructure denoted by } \overrightarrow{f_1} \text{ updated to } v) = v_{\mathrm{new}} \\ (\overrightarrow{R} \text{ with } R_i \text{ replaced by } (a \xmapsto[t]{\overrightarrow{f_0}} v_{\mathrm{new}})) = \overrightarrow{R'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ e_1 = e_2\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R'}\right\}}\ \text{STORE-1}$$

The notation $\overrightarrow{f_0}{+\!+}\overrightarrow{f_1}$ stands for concatenation of paths (i.e. lists of field names or array indices). In Coq, it is written the same, except that the lists are read backwards. Remember from §3.1 that if a is a pointer value of type t, then $a \triangleright \overrightarrow{f}$ denotes the address obtained by following the path $\overrightarrow{f}$ starting from a. Note that it is only well-defined if $\overrightarrow{f}$ is compatible with a's type t, as determined by the assertion field_compatible$(t, \overrightarrow{f}, a)$ (see §3.1).

STORE-1 can be used to prove the Hoare triple for example (a):

$$\frac{\begin{array}{c}\cdots \vdash [\![\texttt{\&(*ctx).rk[i]}]\!] = a + 4 + 4i \ \wedge\ [\![e_2]\!] = v \\ a + 4 + 4i = a \triangleright [\texttt{rk}, i] \qquad [\texttt{rk}, i] = [\texttt{rk}]{+\!+}[i] \qquad \overrightarrow{R}_0 = (a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} \ell) \\ (\ell \text{ with the substructure denoted by } [i] \text{ updated to } v) = (\mathsf{upd\_Znth}\ i\ \ell\ v)\end{array}}{\begin{array}{c}\{\mathsf{PROP}(0 \le i \le 8)\ \mathsf{LOCAL}(\mathsf{temp}\ \texttt{ctx}\ a; \mathsf{temp}\ \texttt{i}\ i)\ \mathsf{SEP}(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} \ell)\} \\ \texttt{(*ctx).rk[i] = } e_2 \\ \{\mathsf{PROP}(0 \le i \le 8)\ \mathsf{LOCAL}(\mathsf{temp}\ \texttt{ctx}\ a; \mathsf{temp}\ \texttt{i}\ i)\ \mathsf{SEP}(a \xmapsto[\texttt{aes\_context}]{[\texttt{rk}]} (\mathsf{upd\_Znth}\ i\ \ell\ v))\}\end{array}}$$

where $(($upd_Znth $i\ \ell\ v)$ updates the ith entry of the list ℓ with the new value v. Note that the expression evaluation $[\![$`&(*ctx).rk[i]`$]\!]$ returns a pointer-arithmetic expression. That's why we need another equality to turn this result into the field address $a \triangleright [\texttt{rk}, i]$.[10]

5.4.3 More automation-friendly specialized load/store rules

Equalities such as $a + 4 + 4i = a \triangleright [\texttt{rk}, i]$ from the previous examples are not easy to prove automatically, because we need all field_compatible side conditions of the field-address operator, and such reasoning results in slow tactics and large proof terms.

We prefer to do this kind of reasoning once and for all, inside a lemma that the tactics can use, instead of doing it every time we apply the rule. The idea is to exploit

[10] One might wonder why the expression evaluation function does not directly return a field address. For `struct` and `union` fields, this could work, but for array indices, it wouldn't, because the field-address operator is only well-defined if the array index is within the array bounds, but adding an integer to a pointer is always defined in CompCert C light, even if dereferencing it might be undefined. So the evaluation function could only use the field-address operator if the array index is within bounds, but it cannot know whether this is the case, because it does not have access to the array size.

the knowledge about the access path that we get by looking at the syntax of the C command. For instance, in example (a) the whole path is visible in the C command.

To do so, we derive special versions of the load and store rules:

$$\text{LOAD-2}\ \frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e]\!] = a \wedge [\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f} \\ \overrightarrow{f} = \overrightarrow{f_0}{+\!+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v') \qquad \mathsf{legal\_nested\_field}\ t\ \overrightarrow{f} \\ (\text{the component in } v' \text{ denoted by } \overrightarrow{f_1}) = v \\ (\overrightarrow{Q} \text{ with the value for } \mathsf{x} \text{ updated to } v) = \overrightarrow{Q'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ \mathsf{x} = e.\overrightarrow{\mathsf{F}}\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q'}\ \mathsf{SEP}\ \overrightarrow{R}\right\}}$$

$$\text{STORE-2}\ \frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e_1]\!] = a \wedge [\![e_2]\!] = v \wedge [\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f} \\ \overrightarrow{f} = \overrightarrow{f_0}{+\!+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v_{\mathrm{old}}) \qquad \mathsf{legal\_nested\_field}\ t\ \overrightarrow{f} \\ (v_{\mathrm{old}} \text{ with the substructure denoted by } \overrightarrow{f_1} \text{ updated to } v) = v_{\mathrm{new}} \\ [\![e_2]\!] = v \qquad (\overrightarrow{R} \text{ with } R_i \text{ replaced by } (a \xmapsto[t]{\overrightarrow{f_0}} v_{\mathrm{new}})) = \overrightarrow{R'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ e_1.\overrightarrow{\mathsf{F}} = e_2\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R'}\right\}}$$

We distinguish $\overrightarrow{\mathsf{F}}$ from $\overrightarrow{f}$: they represent a nested field in a C command and its denotation respectively. A `struct`/`union` field in a C expression is the same as its denotation but an array subscript is an int expression in C while its denotation is a Coq value of type $\mathbb{Z}$. We write this correspondence as $[\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f}$.

Moreover, the assertion $\mathsf{legal\_nested\_field}\ t\ \overrightarrow{f}$ ensures that the path $\overrightarrow{f}$ exists within the type t. In particular, it checks that array indices are within the bounds.

These rules work very well to automate strongest postcondition generation for examples like snippet (a): First, the left-hand side of the assignment is split into a root expression e_1 and a path $\overrightarrow{\mathsf{F}}$, i.e. into `*ctx` and `[rk, i]` in our example. Next, $\&e_1$ and e_2 are evaluated, and $\overrightarrow{f}$ is inferred from a $\overrightarrow{\mathsf{F}}$ such that $[\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f}$ holds. In our example, this results in $[\![\texttt{\&(*ctx)}]\!] = a$ and $[\![\texttt{[rk, i]}]\!] = [\texttt{rk}, i]$. Next, all SEP clauses are searched to find one about a whose path is a prefix of $\overrightarrow{f}$. Note that this must be unique, because the SEP clauses talk about disjoint memory areas, and no two distinct memory areas can be accessed by the same path. Once the SEP clause is found, it's clear how to split $\overrightarrow{f}$ into $\overrightarrow{f_0}{+\!+}\overrightarrow{f_1}$; in our case we have $\overrightarrow{f_0} = [\texttt{rk}]$ and $\overrightarrow{f_1} = [i]$. The SEP clause also gives the value for v_{old}, which is ℓ in our case, and now one has to select the substructure of v_{old} according to the remainder of the path, $\overrightarrow{f_1}$, to obtain the part of v_{old} to be updated. Now, all parameters needed to apply STORE-2 are known, so we obtain a strongest postcondition together with a proof for our claim. Occasionally, some typechecking or array bounds side conditions might not be solved automatically and left open as subgoals to be proven by the user, but this never prevents the tactics from applying the STORE-2 rule, so most of the work is always done automatically.

However, the tactic described above using STORE-2 does not work for the examples (b), (c) and (d): In example (b), the root expression e_1 that it picks is `p`, which

evaluates to $a + 4$, where 4 is the offset of the field rk within the struct aes_context. So, the SEP clauses will be searched for a clause of the form $((a + 4) \mapsto \_)$, which will not be found, because we have only a SEP clause of the form $(a \mapsto \_)$.

As we can see, the problem is that the LOAD-2 and STORE-2 rules assume that the whole access path appears in the source code of the load or store instruction, which is not always the case.

We could resolve this by applying STORE-1 instead (which works for all four examples), but as described before, the proof of the equation $a + 4 + 4i = a \triangleright [\mathsf{rk}, i]$ would be hard to automate.

So we design yet another pair of load and store rules:

$$\text{LOAD-3}\ \frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e]\!] = q \wedge [\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f_b} \\ q = a \triangleright \overrightarrow{f_a} \qquad \overrightarrow{f_a}{+}{+}\overrightarrow{f_b} = \overrightarrow{f_0}{+}{+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v') \\ \mathsf{legal\_nested\_field}\ t\ (\overrightarrow{f_0}{+}{+}\overrightarrow{f_1}) \\ (\text{the component in } v' \text{ denoted by } \overrightarrow{f_1}) = v \\ (\overrightarrow{Q} \text{ with the value for } \mathsf{x} \text{ updated to } v) = \overrightarrow{Q'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ \mathsf{x}{=}e.\overrightarrow{\mathsf{F}}\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q'}\ \mathsf{SEP}\ \overrightarrow{R}\right\}}$$

$$\text{STORE-3}\ \frac{\begin{array}{c}\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R} \vdash [\![\&e_1]\!] = q \wedge [\![e_2]\!] = v \wedge [\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f_b} \\ q = a \triangleright \overrightarrow{f_a} \qquad \overrightarrow{f_a}{+}{+}\overrightarrow{f_b} = \overrightarrow{f_0}{+}{+}\overrightarrow{f_1} \qquad R_i = (a \xmapsto[t]{\overrightarrow{f_0}} v_{\mathrm{old}}) \\ \mathsf{legal\_nested\_field}\ t\ (\overrightarrow{f_0}{+}{+}\overrightarrow{f_1}) \\ (v_{\mathrm{old}} \text{ with the substructure denoted by } \overrightarrow{f_1} \text{ updated to } v) = v_{\mathrm{new}} \\ [\![e_2]\!] = v \qquad (\overrightarrow{R} \text{ with } R_i \text{ replaced by } (a \xmapsto[t]{\overrightarrow{f_0}} v_{\mathrm{new}})) = \overrightarrow{R'}\end{array}}{\Delta \vdash \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R}\right\}\ e_1.\overrightarrow{\mathsf{F}} = e_2\ \left\{\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R'}\right\}}$$

The key insight here is that there are two ways of splitting the access path $\overrightarrow{f}$: The first, $\overrightarrow{f} = \overrightarrow{f_0}{+}{+}\overrightarrow{f_1}$, is imposed by the SEP clause, and the second, $\overrightarrow{f} = \overrightarrow{f_a}{+}{+}\overrightarrow{f_b}$, is given by how much of the path is contained in the value of q vs how much of the path is written out as $\overrightarrow{\mathsf{F}}$ in the C command.

Applying LOAD-3 and STORE-3 can be automated in the same way as LOAD-2 and STORE-2, provided that the root expression e (or e_1, respectively) evaluates to a local variable q for which the LOCAL clauses of the precondition contain an entry of the form (temp q $(a \triangleright \overrightarrow{f_a})$). For situations when this is not the case, we designed a "hint" interaction system:[11] If the tactics fail to prove an equality of the form $q = a \triangleright \overrightarrow{f_a}$, they display an error message asking the user to prove an equality of the form $q =$?a $\triangleright$?f, where q is obtained by computationally evaluating e, and ?a and ?f are Coq evars to be instantiated by the user. That is, the user has to provide a witness that the pointer arithmetic expression can be turned into a field address expression. After proving

[11] The kind of hints we are talking about here are *not* related to Coq's hint databases.

such a hint, the user can retry invoking forward, which invokes the tactics for loads or stores, respectively, and they will use the hint and thus succeed.

So, are LOAD-3 and STORE-3 the ultimate solution which works always? No, for the following two reasons:

First, when $\overrightarrow{f_a}$ is the empty path $[]$, we still have to prove $q = a \triangleright []$, which is not trivially true, because it requires $\mathsf{field\_compatible}_t\ [\,]\ a$ to be proven. Automating this would be possible, but it is easier to do it once and for all in a specialized lemma, and it turns out that we already proved such specialized lemmas: They are exactly the LOAD-2 and STORE-2 rules.

And second, there are cases such as example (d) where we should not blindly trust and use the access path written in the C command: If the tactics match p[-1] with $e_1.\overrightarrow{\mathsf{F}}$, they obtain the path [-1], which is never a valid path, because all arrays in C start with index 0. So, proving the legal_nested_field assertion will fail, because this assertion checks that all array indices are within bounds. In those cases, we have to use LOAD-1 and STORE-1, which do not infer any path fragments from the C code. But their downside is that they are bad at automating the proof of the equation $q = a \triangleright \overrightarrow{f}$. In fact, it only works automatically if the whole $\&e$ (or $\&e_1$, in the store case) evaluates to just a nonaddressable variable q for which we have a LOCAL clause of the form (temp q $(a \triangleright \overrightarrow{f_a})$), such as example (c). In all other cases, we have to resort to the the hint system, and compared to the LOAD-3 and STORE-3 rules, where the required hint is only about the first part of the path, the hint is about the full path here, which is a bit more work for users.

The following table summarizes the advantages and disadvantages of the three store rules with respect to the four examples:

	(a)	(b)	(c)	(d)
	`(*ctx).rk[i]=`	`p=(*ctx).rk; p[i]=`	`p=(*ctx).rk; p++; *p=`	`p=(*ctx).rk+1; p++; p[-1]=`
STORE-1	requires full-path hint	requires full-path hint	requires suitable precondition or full-path hint	requires full-path hint
STORE-2	works automatically	not applicable	not applicable	not applicable
STORE-3	requires $q = a \triangleright []$ proof	requires suitable precondition or root-path hint	requires suitable precondition or root-path hint	not applicable

5.4.4 The final tactic for memory loads/stores

So overall, our final tactic for a memory store of the form $e_1.\overrightarrow{\mathsf{F}} = e_2$ now works as shown in Figure 5 (the tactic for memory loads is similar).

Note that it uses all three STORE rules, even though STORE-1 would be general enough to be applicable in all cases. The only reason to use the other store rules is to overcome the proof automation difficulties. In fact, instead of attempting to use tactics to perform the tricky task of bringing the result of evaluating an expression into the form $a \triangleright \overrightarrow{f}$, we prefer to do this work with lemmas, by having three special-

- Evaluate the whole expression $e_1.\overrightarrow{\mathsf{F}}$ and check if the context contains a user-defined hint on how to bring it into the form $a \triangleright \overrightarrow{f}$.
  - If yes, find a SEP clause about a whose path is a prefix of $\overrightarrow{f}$, and apply STORE-1 and solve its side conditions.
  - Otherwise, evaluate the root expression e_1 and check if the result already has the form $a \triangleright \overrightarrow{f_a}$ or if the context contains a user-defined hint on how to bring it into the form $a \triangleright \overrightarrow{f_a}$.
    - If yes, find $\overrightarrow{f_b}$ such that $[\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f_b}$, and find a SEP clause about a whose path is a prefix of $\overrightarrow{f_a}$++$\overrightarrow{f_b}$, and apply STORE-3 and solve its side conditions.
    - Otherwise, find $\overrightarrow{f}$ such that $[\![\overrightarrow{\mathsf{F}}]\!] = \overrightarrow{f}$, and check if there is a SEP clause about a whose path is a prefix of $\overrightarrow{f}$.
      - If yes, apply STORE-2 and solve its side conditions.
      - Otherwise, fail with an error message containing the form of hints which were not found before.

Fig. 5: Tactic for calculating the strongest postcondition of store commands together with a proof

ized lemmas, and depending on the case, picking the one which is easiest to apply automatically.

Now, one might wonder if this is the end, or whether at some time in the future, we might come across another C code example where these tactics fail. We believe that, except for the deliberately chosen restrictions described in §2.1, there will be no further such C code examples, because LOAD-1 and STORE-1 do not impose any restriction on the form of the expression which denotes the memory location, and if evaluating this expression cannot turn it into something of the form $a \triangleright \overrightarrow{f}$ automatically, the tactics can fall back to the user hint mechanism, allowing to request that the user proves the tricky part in such a way that all the rest can still be solved automatically.

However, the tactics for loads and stores do have one limitation: They only work if the required SEP clause is written as a data_at or field_at assertion. If users create custom separation logic assertions, they will have to unfold them into data_at or field_at form. Automating this would be difficult, because the reason why users create custom separation-logic predicates tend to be domain-specific, so the right strategy to unfold the custom assertions requires domain-specific knowledge as well; this should be left to the user, who may choose to achieve this task by additional domain-specific automation.

5.5 Strongest postcondition of function calls

The effects of C function calls are more complicated than set, load, or store commands. A function call may assign its result to a nonaddressable variable and may

modify the data stored in heap. Fortunately, separation logic allows us to reason about the behaviors of calls and the canonical form allows us to present it concisely.

Consider a function `void swapint(int *x, int *y);` with this specification:

`swapint(x, y)` :
$\Pi(a, b, p, q)$.
$\{\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ p; \mathsf{temp\ y}\ q)\ \mathsf{SEP}(p \underset{\texttt{int}}{\longmapsto} a; q \underset{\texttt{int}}{\longmapsto} b)\}$
$\{\mathsf{PROP}()\ \mathsf{LOCAL}()\ \mathsf{SEP}(p \underset{\texttt{int}}{\longmapsto} b; q \underset{\texttt{int}}{\longmapsto} a)\}$

Assume this specification is in Δ (that is, the fun-spec part of Δ associates this specification with the name `swapint`).

Then consider this function call: `swapint(x, x+2)`, in a context where x is a pointer into an array of at least three consecutive integers. We would have this proof goal:

$$\Delta \vdash \{\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ x)\ \mathsf{SEP}(x \underset{\texttt{int}}{\longmapsto} u;\ x+4 \underset{\texttt{int}}{\longmapsto} v;\ x+8 \underset{\texttt{int}}{\longmapsto} w)\}$$
`swapint(x, x + 2)`
$\{?\mathsf{Post}, \bot, \bot, \bot\}$

The strongest postcondition is,

$$\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ x)\ \mathsf{SEP}(x+4 \underset{\texttt{int}}{\longmapsto} v;\ x \underset{\texttt{int}}{\longmapsto} w;\ x+8 \underset{\texttt{int}}{\longmapsto} u)$$

How this can be generated? First, the user instantiates the parameter (a, b, p, q) with the value $(u, w, x, x+8)$. Then the instantiated specification is:

`swapint(x, y)` :
$\{\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ x; \mathsf{temp\ y}\ (x+8))\ \mathsf{SEP}(x \underset{\texttt{int}}{\longmapsto} u;\ x+8 \underset{\texttt{int}}{\longmapsto} w)\}$
$\{\mathsf{PROP}()\ \mathsf{LOCAL}()\ \mathsf{SEP}(x \underset{\texttt{int}}{\longmapsto} w;\ x+8 \underset{\texttt{int}}{\longmapsto} u)\}$

Second, the LOCAL part of the precondition in this specification is verified automatically (and computationally) by C expression evaluation, i.e.

msubst_eval_expr [temp x x] (x) = x
msubst_eval_expr [temp x x] (x + 2) = $x + 8$

(Since x+2 in C is a pointer-integer add, the semantics of C gives the address $x + 8$.)

Third, for the SEP part, we pick out the precondition of the instantiated specification in the precondition of the proof goal, replace it with the postcondition of the instantiated specification and use the replacement result as the generated postcondition in the proof goal. This "picking out" is a form of "frame inference," and is accomplished by a cancellation tactic.

Fourth, this function call has no return value, so the LOCAL part of the generated postcondition is the same as the precondition.

We mostly automate this process in VST-Floyd, but require our users to manually instantiate the parameters in the specification. The soundness of this process is ensured by the following Hoare rule, SEMAX-CALL-00. We prove it in Coq as a derived rule of Verifiable C.

Lemma semax_call_00

forall $\Delta\ \overrightarrow{Q}\ \overrightarrow{R}\ \overrightarrow{F}\ \overrightarrow{e}\ a$ Pre Post $\overrightarrow{P_{pre}}\ \overrightarrow{R_{pre}}\ \overrightarrow{P_{post}}\ \overrightarrow{R_{post}}\ f$,
$(f(\overrightarrow{\mathsf{x}})\colon \Pi a : A.\ \{\mathsf{Pre}(a)\}\{\mathsf{Post}(a)\}) \in \Delta \to$
Pre a = PROP($\overrightarrow{P_{\mathrm{Pre}}}$) LOCAL(temp_list $\overrightarrow{\mathsf{x}}$ $\overrightarrow{v}$) SEP($\overrightarrow{R_{pre}}$) $\to$
Post a = $\exists$ b: B, PROP($\overrightarrow{P_{\mathrm{Post}}(b)}$) LOCAL() SEP($\overrightarrow{R_{\mathrm{Post}}(b)}$) $\to$
$\overrightarrow{P_{\mathrm{Pre}}} \to$
PROP() LOCAL($\overrightarrow{Q}$) SEP($\overrightarrow{R}$) $\vdash$ tc_expr_list $\overrightarrow{e}$ $\to$
msubst_eval_expr_list $\overrightarrow{e}$ $\overrightarrow{Q}$ = Some $\overrightarrow{v}$ $\to$
Permutation $\overrightarrow{R}$ ($\overrightarrow{R_{\mathrm{Pre}}}$ ++ $\overrightarrow{F}$) $\to$
$\Delta \vdash$ { PROP () LOCAL ($\overrightarrow{Q}$) SEP ($\overrightarrow{R}$) }
$f(\overrightarrow{e})$
{ $\exists$ b: B, PROP($\overrightarrow{P_{\mathrm{Post}}(b)}$) LOCAL($\overrightarrow{Q}$) SEP($\overrightarrow{R_{\mathrm{Post}}(b)}$ ++ $\overrightarrow{F}$), $\bot$, $\bot$, $\bot$ }

A C function may or may not return a value. If it does return a value, the call site may or may not assign that value to a variable. Therefore, we have three cases:

11 The function returns a value, and the call site assigns it to a variable.
01 The function returns a value, but the call site throws it away.
00 The function does not return a value, and (therefore in a well typed C program) the call site does not expect a value.

The example above (with semax_call_00) is of the third kind. The derived lemmas for the other two kinds are similar, and have names ending with 11 and 01.

In the `swapint` example above, the PROP part of the precondition in the specification is empty. If it were not empty, our postcondition generator would check these pure facts.

The postconditions in specifications can be existentially quantified (which does not happen in our example above). Our postcondition generator and the derived Hoare rules do cover those cases. Generally speaking, the generated postcondition is an existentially quantified canonical assertion. When the postcondition in a specification is not quantified, we treat it as quantified over unit type. When the generated postcondition is quantified over unit type, our generator removes the quantifier and presents a quantifier-free version.

Comparing to the cases for set, load or store, the generator for function calls imposes more restrictions on the user. Besides the fact the users need to manually provide values to instantiate specification, users also need to ensure an exact match between specification SEP clauses and part of SEP clauses in the proof goal. Specifically, when handling load or store commands, we do detect $p \xmapsto[\texttt{IntPair}]{\texttt{[fst]}} a$ inside $p \xmapsto[\texttt{IntPair}]{} a, b$. But we choose not to support a similar feature when handling function call because the situation here is much more complicated. For example, the

correspondence between the specification and the precondition in the proof goal may be many-to-one. It is very hard to detect

$$\mathsf{SEP}(p \triangleright \texttt{[fst]} \overset{}{\underset{\texttt{int}}{\longmapsto}} a; p \triangleright \texttt{[snd]} \overset{}{\underset{\texttt{int}}{\longmapsto}} b)$$

inside $p \overset{}{\underset{\texttt{IntPair}}{\longmapsto}} a, b$. We require users to apply related transformations in preconditions first. We provide tactics for these transformation (see §7).

6 Automatic Tactics for Forward-Style Proof

The most important feature of VST-Floyd is its forward proof style. We provide a set of tactics in VST-Floyd to perform forward verification. From one point of view, these tactics help users build proof trees of Hoare triples and automate the routine work. From another point of view, building proofs of Hoare triples using forward tactics is like demonstrating a decorated program from the top down—for example, each call to the tactic forward (see §6.1) is like demonstrating one more C assignment command and one more assertion in a decorated program.

In this section, we introduce the forward tactics which are mostly automatic. They are forward, forward_call, forward_if and forward_while. We will introduce the interactive tactics in the next section.

Remark. Both automatic and interactive tactics are essential in VST-Floyd. On one hand, some proof strategies in verifying a Hoare triple are routine. For example, when the C command in the triple is an sequential composition of multiple commands, we should apply SEMAX-SEQ first. Automatic tactics can easily handle this and let VST-Floyd be more convenient for users. On the other hand, it is helpful for our users to manually manipulate Hoare triples as well. For example, users can apply domain-specific mathematics to simplify the precondition. It is sometimes necessary to unfold some user-defined separation logic predicates to perform later automic tactics. It is also helpful sometimes, to fold some user-defined predicates and get more concise preconditions.

6.1 Forward on set, load or store commands

VST-Floyd offers two tactics, forward and forward_call, to perform forward verification on singleton commands. Specifically, when the first command is a set command, load command or a store command, forward should be applied. When the first command is a function call (with or without return value), forward_call should be applied.

The tactic forward analyzes the first command in a triple and proceeds by forward reasoning to shrink the proof goal. Fig. 6 and 7 shows one example of applying forward. The Coq proof goal on the right side of Fig. 6 is the proof goal before executing forward. The one on the right side of Fig. 7 is the proof goal after executing forward. The decorated programs on the left correspond to the verification process on the right; the shaded lines correspond to the Coq proof goals.

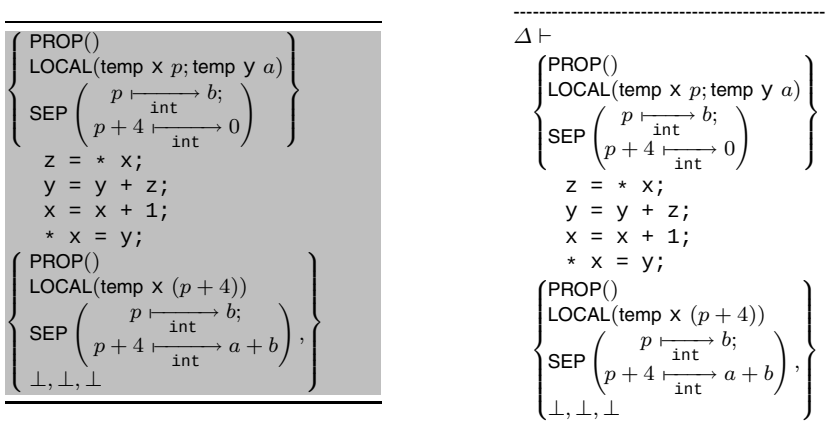


Fig. 6: Before executing forward

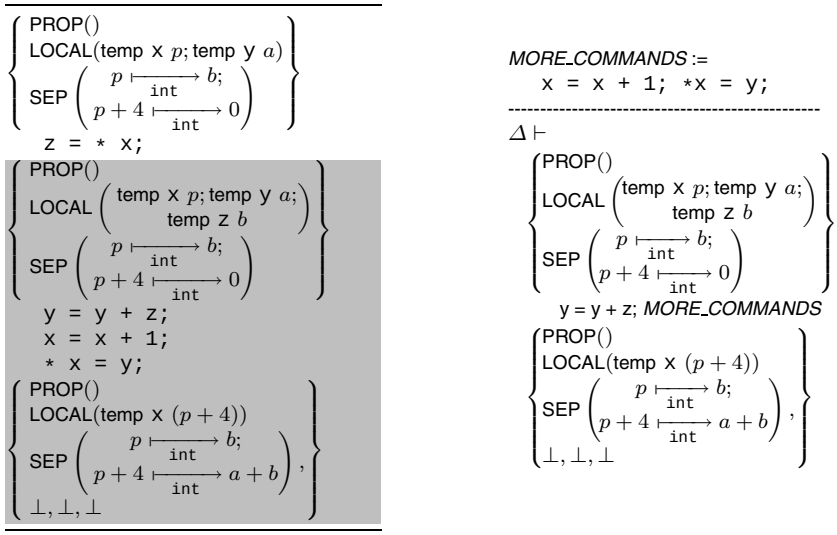


Fig. 7: After executing forward. Floyd has made a local definition *MORE_COMMANDS* so that the proof goal below the line is not cluttered with the entire remainder of the block.

Generally speaking, if the first command in the proof goal is an assignment (set, load or store command) c and the precondition is P, forward will eliminate c and replace the precondition with the strongest postcondition of P and c. Fig. 8 and Fig. 9 demonstrate this process. In Fig. 8, there is more than one command in the original proof goal. Applying forward reduces the proof goal to a new triple in which the new precondition Q is the strongest postcondition of P and c. In Fig. 9, c is the only command in the original proof goal. Applying forward reduces the proof goal to a separation logic entailment, $\Delta \wedge Q \vdash R$.

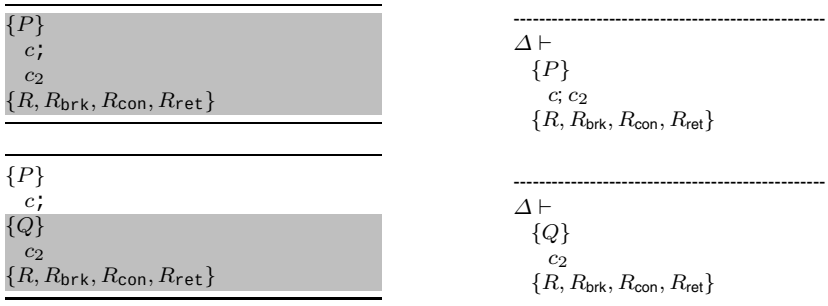


Fig. 8: More than one command

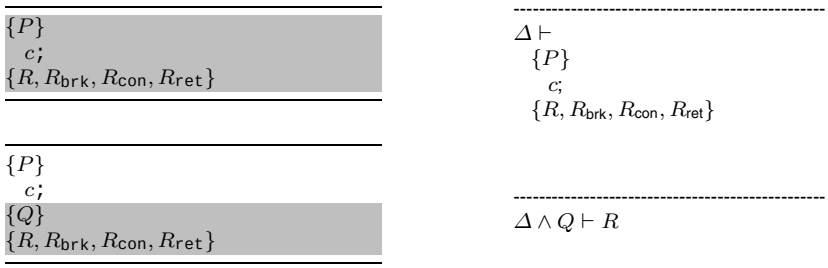


Fig. 9: Only one command

6.1.1 Implementation.

The implementation of forward contains the following steps:

First, forward analyzes the C command in the proof goal. If it has form $(c_1; c_2)$ and c_1 is an assignment, then forward will do

```
eapply semax_seq'
```

If it has is a single assignment command c, then forward will do

```
eapply semax_post'
```

Here, SEMAX_SEQ' and SEMAX_POST' are derived Hoare rules. Coq's eapply tactic creates a unification variable for "floating" variables, in this case the intermediate assertion Q.

$$\text{SEMAX-SEQ'}\frac{\Delta \ \vdash \{P\}\ c_1\ \{Q, \bot, \bot, \bot\} \qquad \Delta \ \vdash \{Q\}\ c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}{\Delta \vdash \{P\}\ c_1; c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}$$

$$\text{SEMAX-POST'}\frac{\Delta \vdash \{P\}\ c\ \{Q, \bot, \bot, \bot\} \qquad Q \vdash R}{\Delta \vdash \{P\}\ c\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}$$

Either way, two proof goals will be generated. The first one has form:

$$\Delta \vdash \{P\}\ c\ \{?Q, \bot, \bot, \bot\}$$

and the second one is either $\Delta \vdash \{?Q\}\ c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}$ or $?Q \vdash R$, respectively. The unification variable $?Q$ is to be filled in later.

In §5, we described our Ltac programs which can generate the strongest post-condition for an assignment and solve the first proof goal. forward calls these Ltac programs and instantiates $?Q$. As a result, the second proof goal, fully instantiated, is presented to users.

Forward requires the original precondition to be in canonical form and ensures that the new precondition is also canonical.

Forward is mostly automatic: most premises and side conditions of rules such as STORE-2 are proved automatically. But sometimes users must prove a side condition (e.g., that some expression evaluation is defined) or to offer some hint (like which data_at or field_at is loaded from or stored to).

6.1.2 Reassociating sequences

The first step in forward is find first command in a sequence of commands. C light formalizes sequential composition as a (deeply embedded) binary syntactic operator. If the proof goal is $\{P\}\ c_1 \cdot (c_2 \cdot c_3)\ \{Q\}$ where c_1 is an assignment statement, forward can use the semax_seq' rule (like HOARE-SEQ in §1.1) to produce the proof goals $\{P\}\ c_1\ \{?U\}$ and $\{?U\}\ c_2 \cdot c_3\ \{Q\}$, where $?U$ is a unification variable to be filled in.

But if the goal is $\{P\}\ (c_1 \cdot c_2) \cdot c_3 \{Q\}$, then forward first reorganizes the sequential composition using this rule:

$$\text{SEQ-ASSOC}\frac{\Delta\ \vdash \{P\}\ c_0 \cdot (c_1 \cdot c_2)\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}{\Delta\ \vdash \{P\}\ (c_0 \cdot c_1) \cdot c_2\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

6.2 Forward on function calls

Forward_call verifies a function call. Its interface and implementation are very similar to forward, except that it takes an argument: the instantiation of the universally quantified parameter of the function specification. We have automated the instantiation of specification parameters.

The following is a specification for `swapint`, which swaps the numbers stored in two different addresses.

```
void swapint(int * x; int * y);
```

`swapint(x, y)`:

$\Pi(a, b, p, q).$

$\{\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp}\ \mathtt{x}\ p; \mathsf{temp}\ \mathtt{y}\ q)\ \mathsf{SEP}(p \underset{\mathtt{int}}{\longmapsto} a; q \underset{\mathtt{int}}{\longmapsto} b)\}$

$\{\mathsf{PROP}()\ \mathsf{LOCAL}()\ \mathsf{SEP}(p \underset{\mathtt{int}}{\longmapsto} b; q \underset{\mathtt{int}}{\longmapsto} a)\}$

Figures 10 and 11 illustrate the effect of running
"`forward_call (`$2a$`, `a^2+1`, `p`, `$p+4$`)`".

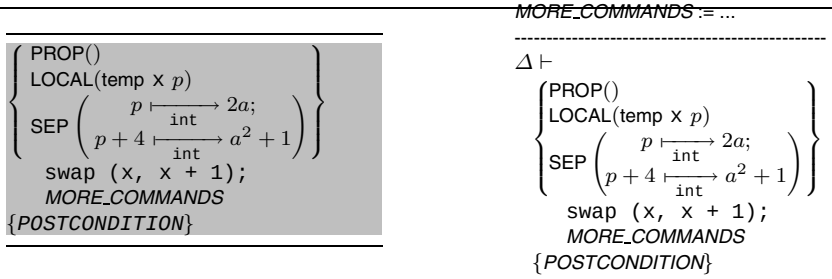


Fig. 10: Before executing forward_call (Fig. 7 explains MORE_COMMANDS).

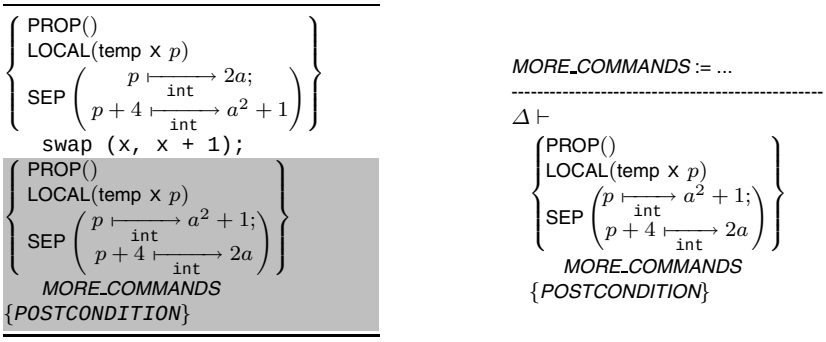


Fig. 11: After executing forward_call

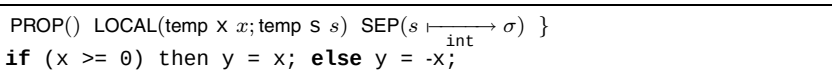
VST-Floyd only requires the postconditions in function specifications to be in existentially quantified canonical form. Thus, the new precondition may have such existentials. We use Intros (see §7.1.2) at the end of forward_call so that the triple left for users has a canonical precondition.

6.3 Forward on if commands

When the first C command in the proof goal is an `if` command, forward_if should be applied to perform forward verification.

In general, after forward proof through two branches of an `if`, one needs to merge the postconditions together. One could say the postcondition is just a disjunction in separation logic, but that just reduces to the problem of eliminating disjunctions and returning to canonical form. So we require the user to provide the joined postcondition as an argument to forward_if; *except* when the `if` command is the only command in the proof goal (e.g., the last command in a block), in which case the postcondition is already concrete (*not* a unification variable), and is therefore already provided.

Consider the following sample program and specification:



```
  t = * s; * s = t + y;
```
$\{\ \mathsf{PROP}()\ \ \mathsf{LOCAL}(\mathsf{temp\ x}\ x; \mathsf{temp\ s}\ s)\ \ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma + |x|),\ \ \bot,\ \ \bot,\ \ \bot\ \}$

The `if` command in this example stores the absolute value of $[\![\mathsf{x}]\!]$ into `y`. We can apply forward_if $\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ x; \mathsf{temp\ s}\ s; \mathsf{temp\ y}\ |x|)\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma)$. Then three subgoals are left for us to prove (see Fig. 12): one for the if-then branch, one for the if-else branch and one for the C commands afterwards.

H: $x \geq 0$

$$\Delta \vdash \left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma) \end{array} \right\}$$

```
  y = x;
```

$$\left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s;\\ \mathsf{temp\ y}\ |x|\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma), \\ \bot, \bot, \bot \end{array} \right\}$$

H: $x \ngeq 0$

$$\Delta \vdash \left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma) \end{array} \right\}$$

```
  y = -x;
```

$$\left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s;\\ \mathsf{temp\ y}\ |x|\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma), \\ \bot, \bot, \bot \end{array} \right\}$$

$$\Delta \vdash \left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s;\\ \mathsf{temp\ y}\ |x|\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma) \end{array} \right\}$$

```
  t = * s;
  * s = t + y;
```

$$\left\{ \begin{array}{l} \mathsf{PROP}() \\ \mathsf{LOCAL}\left(\begin{array}{l}\mathsf{temp\ x}\ x;\\ \mathsf{temp\ s}\ s\end{array}\right) \\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma + |x|), \\ \bot, \bot, \bot \end{array} \right\}$$

Fig. 12: Subgoals after forward_if

Fig. 13 and 14 is a sketch of the effect of forward_if Q in general. One important detail in this tactic interface is how we handle the predicate that "the denotation of b is true/false". Traditionally, $[\![b]\!] = \text{true}$ will be a conjunct of the precondition in the if-then branch. In VST-Floyd, it does not show up in the precondition but appears above the line as an assumption of the whole if-then triple. In our example above, if the precondition of the conditional is

$$\mathsf{PROP}()\ \mathsf{LOCAL}(\mathsf{temp\ x}\ x; \mathsf{temp\ s}\ s)\ \mathsf{SEP}(s \underset{\mathtt{int}}{\longmapsto} \sigma)$$

and b is `x >= 0`. Thus $x \geq 0$ will an assumption of the if-then triple. Similarly, $x \ngeq 0$ will be an assumption of the if-else triple. Treating this testing result as a Coq assumption about values, instead of a Hoare logic precondition about expressions, is very convenient in verifying real C programs.

Implementation. In "forward_if Q", we first apply SEMAX-SEQ to split the proof goal into two. The first one will be handled by forward_if and the second one, a triple for the rest of program is directly left to the users.

The implementation of forward_if is based on the following auxiliary Hoare rule.

$$\textsc{semax-if'}\ \frac{\begin{array}{c} P \vdash \text{tc\_expr}(\Delta, b) \\ P \vdash [\![b]\!] = v \\ \text{If } v = \text{true, then } \Delta \vdash \{P\}\ c_1\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\} \\ \text{If } v = \text{false, then } \Delta \vdash \{P\}\ c_2\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\} \end{array}}{\Delta\ \vdash \{P\}\ \mathsf{if}\ (b)\ c_1\ \mathsf{else}\ c_2\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

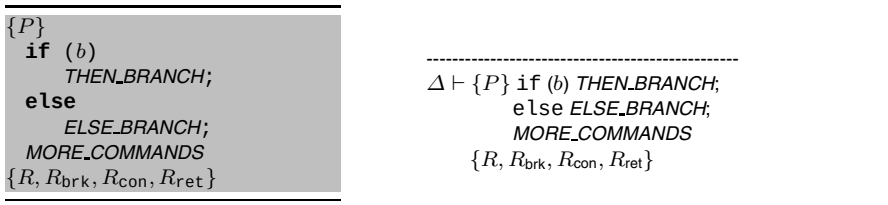


Fig. 13: More than one if command—before forward_if

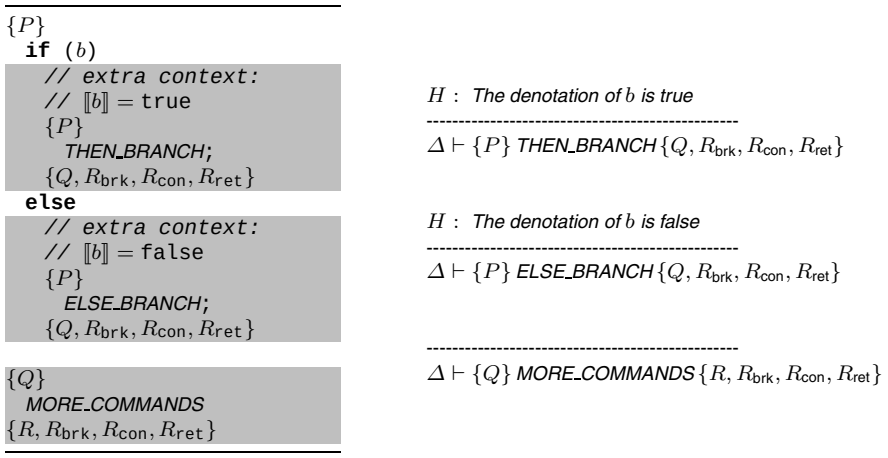


Fig. 14: More than one if command—before forward_if

We do eapply semax_if' first in forward_if. Four subgoals are generated. The first one is typechecking; it will usually be solved automatically. The second is C expression evaluation, which solves by computation and v will be instantiated (see §5.1) if the precondition is in the canonical form. The last two proof goals (then-clause Hoare triple, else-clause Hoare triple) will be presented to users.

6.4 Forward proof on loops

Recall that C light unifies different loops in C into the form `loop(`c_i`)` c. In this general loop command, c is the loop body and c_i is the increment command. Specifically, "`while (`b`)` c" is defined as

`loop(;) {if (`b`) /*skip*/; else break;` c`}`

and "`for (`c_0`;` b`;` c_i`)` c" is defined as

c_0`; loop(`c_i`) {if (`b`) /*skip*/; else break;` c`}`.

Verifiable C offers a primary Hoare rule for general loops.

$$\text{SEMAX-LOOP}\ \frac{\begin{array}{l}\Delta\ \vdash \{P\}\ c\ \{P', Q, P', Q_{\text{ret}}\} \\ \Delta\ \vdash \{P'\}\ c_i\ \{P, \bot, \bot, Q_{\text{ret}}\}\end{array}}{\Delta\ \vdash \{P\}\ \mathsf{loop}\ (c_i)\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\}}$$

This rule is already pretty easy to use, compared to other primary Hoare rules in Verifiable C. There is no expression evaluation or typechecking criterion involved. So we do not provide forward tactics for general loops. Users can apply this rule directly. To apply this rule in forward verification, users need to provide two loop invariants, one before incremental step (P') and one after incremental step (P), and one postcondition (Q).

However, it's inconvenient to provide three assertions P', P, Q when there are common special cases where only one is needed. For `while` loops, the two loop invariants are identical. For `while` loops without `break` command inside, the postcondition is just a conjunction of the loop invariant and a side condition that the loop condition is false. Therefore, we provide a tactic forward_while for simple while loops without break statements.

We also have a special-purpose for-loop tactic, forward_for_simple_bound, to handle loops of the form `for(...; i<`E`; i++)`.

7 Interactive Tactics for Forward-Style Proof

In the previous section, we introduced forward tactics in VST-Floyd which perform forward verification. But sometimes those tactics do not connect well to each other. All forward tactics assume that the precondition in the proof goal is in canonical form, but some tactics' postconditions (e.g., forward_call), and typical loop invariants, are in existentially quantified canonical form.

Another problem is SEP clause reorganization. The forward for load and store commands only detects predicates with form $p \underset{t}{\longmapsto} v$ and $p \underset{t}{\overset{\vec{f}}{\longmapsto}} v$ in SEP clauses but cannot handle user-defined predicates. The forward_call tactic requires an exact match between specification SEP clauses and a subset of the SEP clauses in the proof goal. Users must sometimes rewrite the precondition of the proof goal so that forward can work.

In this section, we introduce the interactive tactics in VST-Floyd to manipulate assertions and triples. These tactics are the glue code of forward tactics.

7.1 Intros

Intros is the basic tactic to extract PROP clauses in preconditions, to pull out existentially quantified variables and to improve the arrangement of canonical preconditions.

7.1.1 Extract PROP *clauses*

The first effect of Intros is extracting PROP clauses. Fig. 15 is such an example. The right side of it shows the Coq proof goals before and after applying Intros: two propositions $a \geq 0$ and $b \geq 0$ are dragged above the line and become assumptions of the whole triple. The corresponding decorated program says intuitively: "from this point

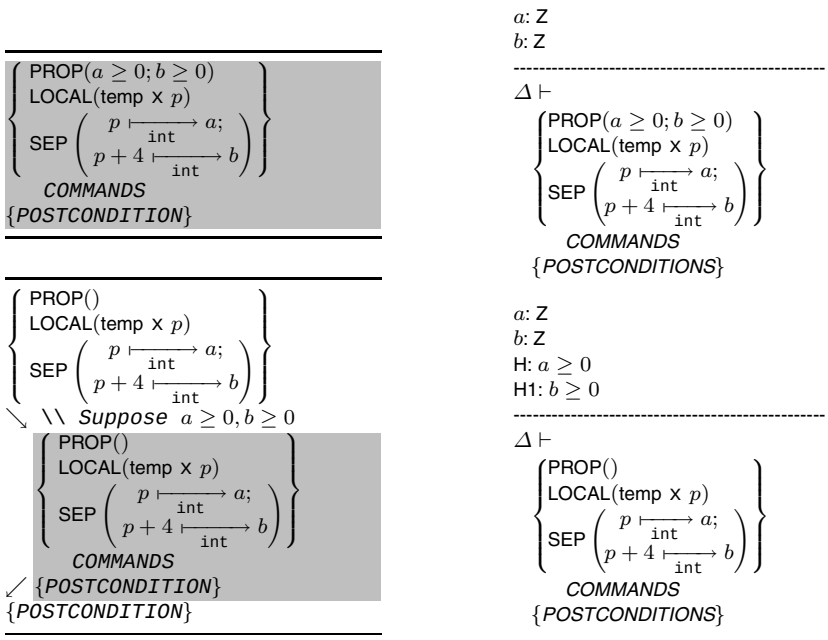


Fig. 15: Example: Intros

to the end of current block, let's assume $a \geq 0$ and $b \geq 0$". Such transformations are sound because of the following proof rule.

$$\text{EXTRACT-PROP}\ \frac{\text{If } P \text{ then } \Delta \vdash \left\{\text{PROP}\ \overrightarrow{P_A}\ \text{LOCAL}\ \overrightarrow{Q_A}\ \text{SEP}\ \overrightarrow{R_A}\right\}\ c\ \{Post\}}{\Delta \vdash \left\{\text{PROP}\ (P; \overrightarrow{P_A})\ \text{LOCAL}\ \overrightarrow{Q_A}\ \text{SEP}\ \overrightarrow{R_A}\right\}\ c\ \{Post\}}$$

It is a very practical proof strategy in VST-Floyd to pull propositions from the precondition into Coq assumptions. It enables users to apply domain-specific mathematics in a more flexible environment, i.e. the Coq's proof mode. For example, $a \geq 0$, $b \geq 0$ and $a + b = 0$ together tell us $a = b = 0$. This can be proved directly by Coq's default solver for linear programming omega. Moreover, if assumptions like $a = 0$ is above the line, Coq tactics like subst can be applied to further simplify the proof goal.

7.1.2 Extract existentials

Intros also extracts existentially quantified variables. If the proof goal is a Hoare triple, Intros x y moves existential quantifiers in the precondition above the line, introducing Coq variables x, y.

One or more arguments after Intros indicate the name of the new Coq variables. Fig. 16 is a tiny example. Intros i will transform the proof goal on the top to the one

on the bottom. Intros also finds existential quantifiers inside a SEP clause (see Fig. 17).

Intros works (soundly) by applying EXTRACT-EXISTS, proved in Coq:

$$\text{EXTRACT-EXISTS}\ \frac{\text{Forall } x.\ (\Delta \vdash \{P(x)\}\ c\ \{Q\})}{\Delta \vdash \{\exists x : A.\ P(x)\}\ c\ \{Q\}}$$

Moving existential quantifiers out of one SEP conjunct is sound due to the commutativity between existential quantifier and separating conjunction: $(\exists a.P(a)) * Q \dashv\vdash \exists a.P(a) * Q$. We prove the following derived rule for canonical assertions. More specifically, our Ltac program always moves an existentially quantified SEP conjunct to the beginning first, then apply this rule.

$$\mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}(\exists x.R_0(x); \overrightarrow{R}) \dashv\vdash \exists x.\ \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}(R_0(x); \overrightarrow{R})$$

7.1.3 Flatten the SEP *clauses*

The other effect of Intros is flattening the SEP clauses. In the original precondition, there is only one SEP clause, but the clause itself is a separating conjunction. Intros splits it into two SEP clauses. This transformation is sound because of the commutativity and associativity of separating conjunction. Fig. 18 shows an example.

7.2 Introduce more propositions in the context

We have shown that Intros can extract propositions in PROP clauses. Besides that, VST-Floyd provides another tactic assert_PROP which adds an proposition P_0 into the Coq assumption if it is derivable from the precondition.

Fig. 19 shows an example of assert_PROP. Here, $(\mathsf{isptr}\ p)$ says p is a pointer value and it is not null. Applying assert_PROP $(\mathsf{isptr}\ p)$ adds this proposition to the Coq assumption list (see the change from the proof goal on the top right to the one on the bottom right). Users are responsible for the soundness of this operation. In this case, we need to prove a separation logic entailerment: precondition implies $(\mathsf{isptr}\ p)$. We know this is true because some data is store at address p. VST-Floyd offers tactics to solve entailments (semi)automatically (see §9). Here, we can apply entailer! to solve this sidecondition.

7.3 Manipulating separating conjunctions in SEP clauses

We provide gather_SEP and replace_SEP to manipulate SEP clauses in preconditions or in the left sides of entailments.

For example, suppose the proof goal is a triple (or an entailment, resp.) whose precondition (or left side, resp) is

$$\mathsf{PROP}()\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}(a; b; c; d; e; f; g; h; i; j)$$

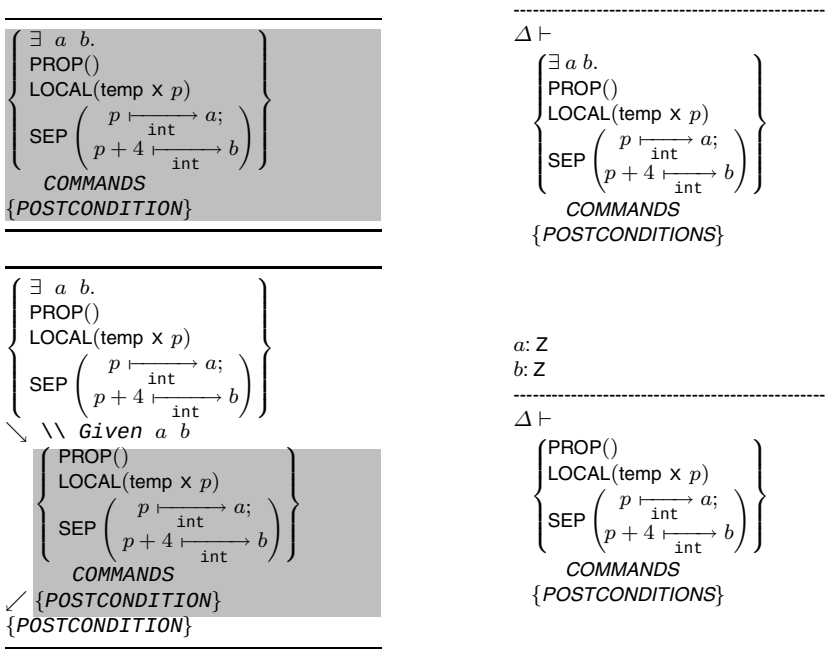


Fig. 16: Example: Intros a b

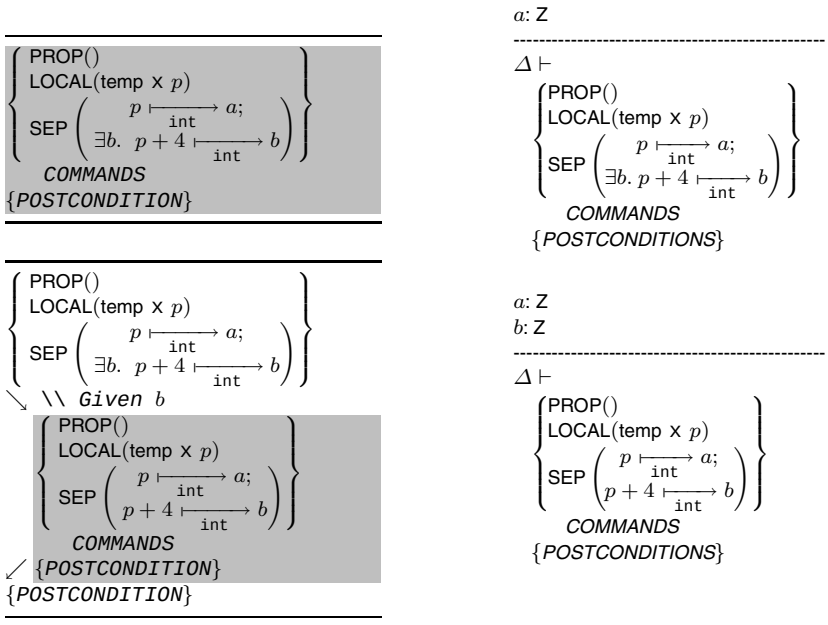


Fig. 17: Example: Intros b

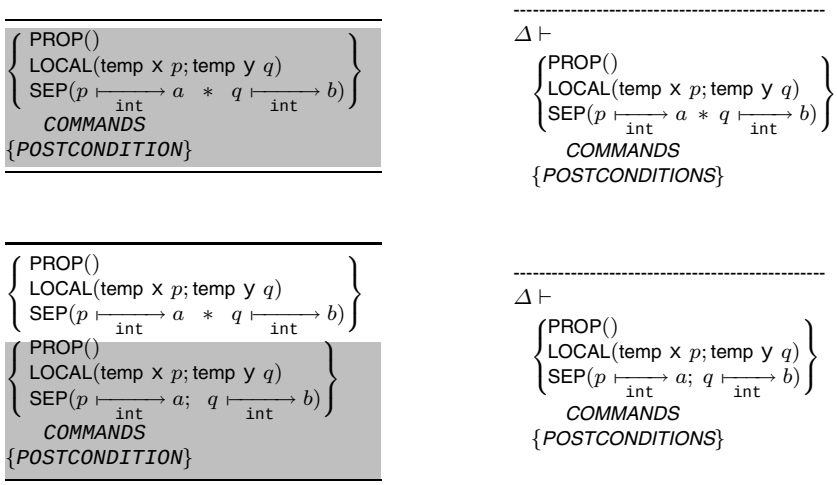


Fig. 18: Example: Intros

gather_SEP i j k will bring the ith, jth, and kth items to the front of the SEP list and conjoin them into a single element.

gather_SEP 5 results in PROP() LOCAL $\overrightarrow{Q}$ SEP$(f; a; b; c; d; e; g; h; i; j)$.

gather_SEP 1 3 results in PROP() LOCAL $\overrightarrow{Q}$ SEP$(b * d; a; c; e; f; g; h; i; j)$.

gather_SEP 3 1 results in PROP() LOCAL $\overrightarrow{Q}$ SEP$(d * b; a; c; e; f; g; h; i; j)$.

replace_SEP i R will replace item #i with predicate R.

replace_SEP 5 R results in PROP() LOCAL $\overrightarrow{Q}$ SEP$(a; b; c; d; e; R; g; h; i; j)$ and a proof subgoal:

$$\Delta \wedge \text{PROP}() \text{ LOCAL } \overrightarrow{Q} \text{ SEP}(f) \vdash R$$

These tactics are very useful to handle user-defined predicates. In §4, we showed a user-defined predicate list which describes integer linked lists. Here, we use a simple version of it to demonstrate how the tactics above can be used.

$$\text{list2}(p, a, b) ::= \exists q.\ p \underset{\text{IntList}}{\longmapsto} a, q \ * \ q \underset{\text{IntList}}{\longmapsto} b, \text{null}$$

This predicate list2 describes a 2-element linked list.

Now, suppose we start from the following proof goal:

$\Delta \vdash \{\text{PROP}()\text{LOCAL}(\text{temp x } p)\text{SEP}(\text{list2}(p, a, b))\}$ ... $\{...\}$

To unfold this user defined predicate, we can do "unfold list2; Intros q":

q: val

$\Delta \vdash \{\text{PROP}()\text{LOCAL}(\text{temp x } p)\text{SEP}(p \underset{\texttt{IntList}}{\longmapsto} a, q; q \underset{\texttt{IntList}}{\longmapsto} b, \text{null})\}$ ... $\{...\}$

If we want to refold this definition, we can first apply "gather_SEP 0 1":

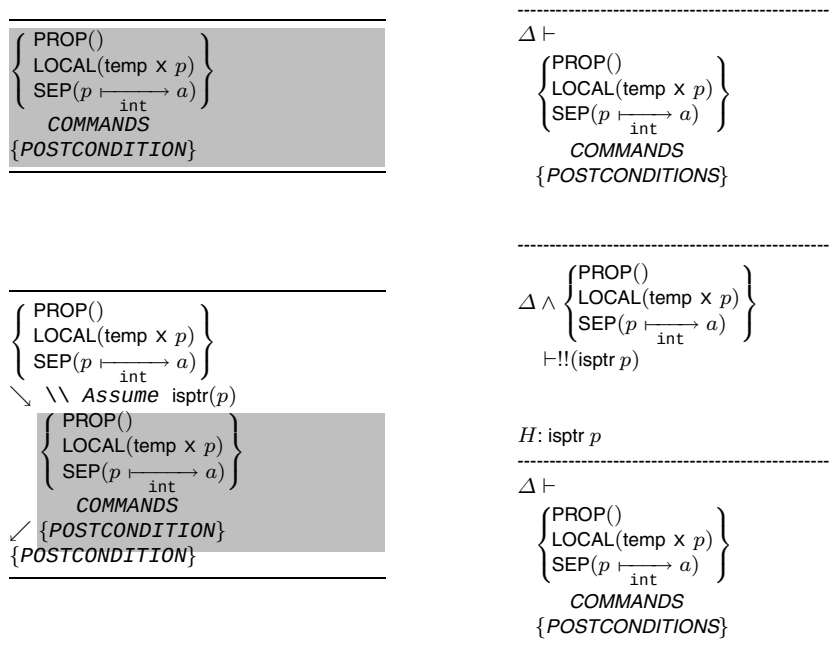


Fig. 19: Example: assert_PROP (isptr p)

q: val

$\Delta \vdash \{\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ x}\ p)\mathsf{SEP}(p \overset{\texttt{IntList}}{\longmapsto} a, q\ *\ q \overset{\texttt{IntList}}{\longmapsto} b, \mathsf{null})\}$ ... $\{...\}$

then tactic "replace_SEP 1 (list2(p, a, b))" can change the proof goal back to the original one. Here, we will be left to prove a subgoal:

q: val

$\Delta \wedge \mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ x}\ p)\mathsf{SEP}(p \overset{\texttt{IntList}}{\longmapsto} a, q\ *\ q \overset{\texttt{IntList}}{\longmapsto} b, \mathsf{null}) \vdash \mathsf{list2}(p, a, b)$

To solve this proof goal, we can do "unfold list2; Exists q; entailer!". We will introduce the tactics for solving entailments in §9.

A more convenient tactic, in place of gather_SEP/replace_SEP is sep_apply f, where f is any lemma of the form $P_1 * \cdots * P_n \vdash Q$; it gathers the P_i from wherever they appear in the left-hand side of the entailment, and replaces them with Q.

8 Structural Proof Rule in Forward-Style Proof

We have shown our tactic library for forward style proof in the previous sections (§6 and §7). Besides that, VST-Floyd also provides support for some structural proofs.

The most useful structural rules are the sequence rule and the frame rule. We introduce their corresponding proof rules and tactics in VST.

8.1 Reorganizing Sequential Composition and Applying Sequence Rule

Suppose the user writes a program,

$$c_1;\ c_2;\ c_3;\ c_4;\ \backslash\!*\ \text{blank line}\ *\!\backslash\ c_5;\ c_6;\ c_7;$$

Here, the blank line in the middle means that c_1, c_2, c_3 and c_4 are for one subtask and c_5, c_6 and c_7 are for another subtask. It is natural for our user to verify these two segments of code separately and then achieve the correctness of the whole program by SEMAX-SEQ. In other words, we want to prove the triple on the top right of Fig. 20 by decomposing it into two triples on the bottom right. The left column of Fig. 20 shows the corresponding decorated program.

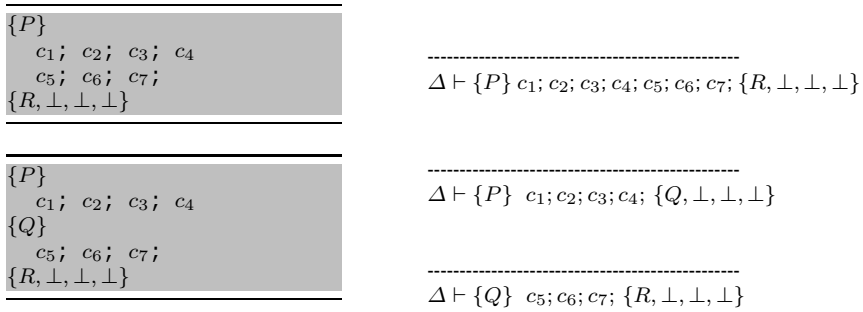


Fig. 20: Proof by subtask decomposition

Recall that the sequential composition of C commands is formalized as a binary oparator in CompCert Clight (see §6.1.2). That means: SEMAX-SEQ cannot be applied directly and we must refactor $c_1 \cdot (c_2 \cdot (c_3 \cdot (c_4 \cdot (c_5 \cdot (c_6 \cdot c_7)))))$ into $(c_1 \cdot (c_2 \cdot (c_3 \cdot c_4))) \cdot (c_5 \cdot (c_6 \cdot c_7))$. The SEQ-ASSOC rule (§sec:id-1st-cmd) cannot do it, since it does not come with a congruence rule (for complicated reasons related to step-indexing of our underlying model).

To reassociate a block, we define a Coq function unfold_Ssequence to turn a C command into a list of commands; it flattens the syntax tree of sequential composition:

$$\begin{aligned}
&\mathsf{unfold_Ssequence}(c) &&::= [c] && \text{if } c \text{ is not a sequential composition}\\
&\mathsf{unfold_Ssequence}\{c_1; c_2\} &&::= \mathsf{unfold_Ssequence}(c1) {+}{+} \mathsf{unfold_Ssequence}(c2)
\end{aligned}$$

where $++$ is list concatenation. We prove the Hoare rule,

$$\text{SEMAX-UNFOLD-SEQ}\ \frac{\mathsf{unfold\_Ssequence}(c_1) = \mathsf{unfold\_Ssequence}(c_2) \qquad \Delta\ \vdash \{P\}\ c_1\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}{\Delta \vdash \{P\}\ c_2\ \{R, R_{\text{brk}}, R_{\text{con}}, R_{\text{ret}}\}}$$

This rule says that we can always reorganize the tree structure of sequential composition as long as the flattened version is unchanged.

To accomplish the decomposition that we described in the beginning of this subsection, the user can eapply SEMAX-UNFOLD-SEQ and SEMAX-SEQ first, then apply two proved Hoare triples for each subtasks.

8.2 Tactical support for flexible framing

VST-Floyd's proof tactics for function calls, loads, and stores automatically include identification of separation logic frames using a weak form of abduction [9], sometimes exploiting programmer hints such as the instantiation witnesses in forward_call. Efficiency of these tactics (and of explicit calls to cancel or entailer typically found in the vicinity of these forward steps) depends directly on the size of the SEP clause. This suggests that proof-checking could be made faster by the liberal use of the frame rule (i.e. SEMAX-FRAME)—aside from the fact that the frame rule enables users to prove triples with more concise pre/postconditions. Unfortunately, as commonly phrased, SEMAX-FRAME does not interact well with the granularity at which forward operates.

$$\text{SEMAX-FRAME}\ \frac{\Delta \vdash \{P\}\ c\ \{Q, Q_{\text{brk}}, Q_{\text{con}}, Q_{\text{ret}}\} \qquad \text{closed\_wrt\_modvars}(c, F)}{\Delta \vdash \{F * P\}\ c\ \{F * Q, F * Q_{\text{brk}}, F * Q_{\text{con}}, F * Q_{\text{ret}}\}}$$

As an example, consider a proof goal of the form $\{P_1 * P_2 * P_3\}\ c_1; c_2; c_3\ \{?Q\}$ with derivable triples $\{P_i\}\ c_i\ \{Q_i\}$. In many (but not all) cases, the first forward-step through this goal implicitly frames off $P_2 * P_3$, steps through c_1, and leaves the proof goal $\{Q_1 * P_2 * P_3\}\ c_2; c_3\ \{?Q\}$. Then, the next two forwards frame off $Q_1 * P_3$ and $Q_1 * Q_2$, respectively.

Matching this behaviour by explicitly wrapping each individual forward step in a maximally precise application of SEMAX-FRAME would rapidly pollute the proof script with administrative clutter. On the other hand, identifying upfront the number of forward steps for which each individual clause in SEP is to be framed off requires inspection of *MORE_COMMANDS*, and SEMAX-FRAME in fact does not allow the specification of such information.

Our solution to this challenge is to rephrase framing as the transfer of clauses between the regular SEP compartment of canonical assertions and specialized compartments that are inaccessible to the forward tactics. A user may invoke the tactic freeze $L\ n$ to bundle a list L of clauses from SEP (where members are identified positionally, much as in the tactics described in Section 7.3) into a named entity $\mathsf{FR}(n)$ that is opaque to the symbolic execution tactics, cancellation, and entailment tactic. Later, she may selectively dissolve such a *freezer* using the tactic thaw n. Each freezer constitutes just another SEP clause, so multiple freezers may coexist or be nested inside one another. Hence, a flexible programmer-controlled framing mechanism is obtained, the use of which can be balanced against the performance of the remaining automation tactics. In typical cases, we have experienced speed-ups of approximately 30%, depending on the number of unfrozen clauses and the complexity of the frozen ones. Future work may seek to improve the declarativeness of freezer management and explore potential connections to the theory of ramification [18].

9 Solving entailments

When verifying a Hoare triple, users need to prove entailments with the form $\Delta \wedge A \vdash B$ in some particular circumstances. For example, at the end of a basic block, the forward tactic will leave an entailment as a subgoal (§6.1). To handle a `while` loop, one must first prove such an entailment: the precondition implies the loop invariant (§6.4). Users of VST-floyd can use replace_SEP to better organize the precondition in the proof goal (§7.3); users are required to prove entailments for the soundness of such replacement.

We provide two tactics Intros and Exists for manipulating quantifiers and we (semi)automate the proofs of quantifier-free entailments using an Ltac program called entailer.

9.1 Intros

If the proof goal is a separation logic entailment, Intros moves existential quantifiers on the left side of entailment to the assumptions of Coq proof goal—much like the way it works on a Hoare triple (§7.1). The soundness of this tactic is based on the following proof rules.

$$\text{EXTRACT-PROP-ENTAIL}\ \frac{\text{If } P \text{ then } \Delta \wedge \mathsf{PROP}\ \overrightarrow{P_A}\ \mathsf{LOCAL}\ \overrightarrow{Q_A}\ \mathsf{SEP}\ \overrightarrow{R}_A \vdash B}{\Delta \wedge \mathsf{PROP}\ (P; \overrightarrow{P_A})\ \mathsf{LOCAL}\ \overrightarrow{Q_A}\ \mathsf{SEP}\ \overrightarrow{R}_A \vdash B}$$

$$\text{EXTRACT-EXISTS-ENTAIL}\ \frac{\text{Forall } x.\ \Delta \wedge P(x) \vdash Q}{(\Delta\ \wedge\ \exists x.\ P(x))\ \vdash\ Q}$$

The context Δ specifies types of local variables and specifications of global functions. The proposition typecheck_environ Δ ρ asserts that every local variable in the environment ρ has a value consistent with the type specified in Δ.

Recall that these rules regard *lifted* assertions, that is, $(\Delta\ \wedge\ \exists x.\ P(x))\ \vdash\ Q$ means $\forall \rho.\ ((\Delta\rho\ \wedge\ \exists x.\ P(x)\rho)\ \vdash\ Q\rho)$ and in such a context, $\Delta\rho$ should be taken to mean, typecheck_environ Δ ρ.

Specifications of global functions are irrelevant to entailments (they are used at function-call Hoare triples and in a Hoare rule for copying a function address to a variable). This aspect of Δ is not tested by typecheck_environ.

9.2 Exists

Exists is the dual of Intros. It instantiates an existentially quantified variable on the right side of an entailment. Fig. 21 shows an example of Exists.

It is sound because of the following proof rule:

$$\text{EXP-RIGHT}\ \frac{P \vdash Q(a)}{P \vdash \exists x.\ Q(x)}$$

In short, Intros and Exists are very similar to Coq's tactics intros and exists, but they perform transformation in the object language (separation logic entailment) instead of in the metalanguage (Coq).

$$\Delta \wedge \left\{\begin{array}{l}\mathsf{PROP}() \\ \mathsf{LOCAL}(\mathsf{temp}\ \mathtt{x}\ p) \\ \mathsf{SEP}(p \xmapsto[\mathtt{int}]{} 4)\end{array}\right\} \vdash \left\{\begin{array}{l}\exists a.\mathsf{PROP}() \\ \quad \mathsf{LOCAL}(\mathsf{temp}\ \mathtt{x}\ p) \\ \quad \mathsf{SEP}(p \xmapsto[\mathtt{int}]{} a^2)\end{array}\right\}$$

$$\Delta \wedge \left\{\begin{array}{l}\mathsf{PROP}() \\ \mathsf{LOCAL}(\mathsf{temp}\ \mathtt{x}\ p) \\ \mathsf{SEP}(p \xmapsto[\mathtt{int}]{} 4)\end{array}\right\} \vdash \left\{\begin{array}{l}\mathsf{PROP}() \\ \mathsf{LOCAL}(\mathsf{temp}\ \mathtt{x}\ p) \\ \mathsf{SEP}(p \xmapsto[\mathtt{int}]{} 2^2)\end{array}\right\}$$

Fig. 21: Example: Exists 2

9.3 Entailer

The tactic entailer does not always prove the entire entailment, but simplifies it by proving those parts that can be automated, without turning provable goals into unprovable goals. It may leave a residual goal.

From $\Delta \wedge \mathsf{PROP}\ \overrightarrow{P_A}\ \mathsf{LOCAL}\ \overrightarrow{Q_A}\ \mathsf{SEP}\ \overrightarrow{R_A} \vdash \mathsf{PROP}\ \overrightarrow{P_B}\ \mathsf{LOCAL}\ \overrightarrow{Q_B}\ \mathsf{SEP}\ \overrightarrow{R_B}$, entailer proceeds in the following steps:

1. Clear from "above the line" any hypotheses known to be irrelevant. For example, in Δ the mapping of function names to function specs is never useful in solving these entailments (it is only used in forward_call), so we prune this mapping from Δ. This has a surprisingly good effect on Coq's execution time in performing the rest of the proof steps.
2. Move the propositions in $\overrightarrow{P_A}$ above the line, by repeatedly applying EXTRACT-PROP-ENTAIL. In the process, apply Verifiable-C-specific lemmas to each P. For example, C-language expression-semantics in P may be simplified to mathematical propositions such as $x < y$. If P is $x = E$, substitute x globally.
3. Go from an entailment in the lifted separation logic (predicates on stacks and heaps) to the unlifted logic (predicates on heaps). We can do this because, unlike a Hoare triple, both A and B are on the same local-variable state. That is, $A \vdash B$ is definitionally equal to $\forall \rho : stack.\ A\rho \vdash B\rho$, so simply intro ρ.
4. Then the elements of $\overrightarrow{Q_A}$ can be moved above the line one at a time, by lemmas such as this one:

   Lemma lower_one_temp:
   forall t $\rho\ \Delta\ \overrightarrow{P}\ i\ v\ \overrightarrow{Q}\ \overrightarrow{R}\ S$,
   $\Delta(\mathtt{i}) = t\ \rightarrow$
   (tc_val $t\ v\ \rightarrow\ [\![\mathtt{i}]\!]\rho = v\ \rightarrow\ (\Delta \wedge \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ \overrightarrow{Q}\ \mathsf{SEP}\ \overrightarrow{R})\ \rho \vdash S) \rightarrow$
   $(\Delta \wedge \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ (\mathsf{temp}\ \mathtt{i}\ v;\ \overrightarrow{Q})\ \mathsf{SEP}\ \overrightarrow{R})\ \rho \vdash S$.

   This matches a *lifted entailment* goal of the form
   $\Delta \wedge \mathsf{PROP}\ \overrightarrow{P}\ \mathsf{LOCAL}\ (\mathsf{temp}\ \mathtt{i}\ v;\ \overrightarrow{Q}) \vdash S'$ where $S' =_\beta S\rho$; that lifted entailment is β-equivalent to the conclusion of the lemma for some arbitrary ρ. The entailer applies this lemma, leaving two subgoals. The first subgoal looks up the C-language type of i in Δ and solves easily by computation.

   The entailer solves the second subgoal by first performing performs two intros, putting tc_val $t\ v$ and $[\![i]\!]\rho = v$ above the line. The former of these says that v is a well-typed value for C-language type t.

The semantics in B of LOCAL(temp i v; Q)ρ is just $[\![\mathtt{i}]\!]\rho = v \wedge \mathsf{LOCAL}\, Q\, \rho$. So, after $\overrightarrow{Q_A}$ is processed, all the temp conjuncts in $\overrightarrow{Q_B}$ are now proved directly from assumptions above the line. The process is analogous for lvar$s$ and gvar$s$.

At this point we may clear from the proof goal anything dependent on ρ. But the tc_val t v hypotheses remain above the line, for use in the remaining steps (and in user proofs of residual goals). For example, if t is C's unsigned-char type, this assumption guarantees that v is indeed an integer (not a pointer or float), and is between 0 and 255.

5. What remains takes the form, $\bigstar\overrightarrow{R_A} \vdash \bigwedge \overrightarrow{P_B} \wedge \bigstar\overrightarrow{R_B}$, where $\overrightarrow{P_B}$ is a list of pure propositions, and $\overrightarrow{R_A}, \overrightarrow{R_B}$ are lists of spatial predicates. For each element of R_A, deduce certain "standard" pure propositions and put these propositions above the line. For example, $p \mapsto v$ implies isptr(p), that p is a pointer (and not null). More interesting, data_at guarantees that the length of each (sub)array *contents* matches the size in the declared C-language type. For each kind of spatial predicate that we might find as a conjunct of R_A, we look for the "standard" proposition in a "Hint database" called saturate_local.
6. Now perform various simplifications and rewrites specific to our C type system and C-language comparison expressions, then general rewriting such as $0 \cdot n = 0$.
7. The next step depends on whether the user has invoked the entailer tactic, which guarantees not to turn a provable goal into an unprovable goal; or entailer! which is a bit more aggressive and efficient.
   - entailer eliminates each element of P_B that is provable with Coq's auto tactic; then repeats from step 6 as long as progress is made.
   - entailer! splits the proof-goal $\overrightarrow{R_A} \vdash \overrightarrow{P_B} \wedge \overrightarrow{R_B}$ into two goals, $\overrightarrow{P_B}$ and $\overrightarrow{R_A} \vdash \overrightarrow{R_B}$. It then tries to prove each of the P_B by auto, compute, omega, reflexivity, leaving a conjunction of the unproved subgoals as a residue for the user. It uses the cancel tactic (see below) to cancel terms that appear both in R_A and R_B, leaving the residue (if any) as a proof goal for the user.

Our cancel tactic takes a goal $R_A \vdash R_B$, where R_B may contain a unification variable if the purpose is *frame inference* in a forward_call proof. Phase one carefully avoids unifying the unification variable (if present), and just walks through each conjunct of R_A finding a corresponding conjunct of R_B. The correspondence need not be exact; there is a Coq "Hint database" of cancellation lemmas, of the form $R_1 \vdash R_2$. Suppose R_A is the first conjunct of the left-hand-side of the entailment. After matching R_1 to R_A (instantiating the lemma), we find the R_2 that matches the instantiated R_B, and do some directed AC rewriting to pull R_2 to the front of the conjunction. Then the lemma $(R_1 \vdash R_2) \to (R_A \vdash R_B) \to (R_1 * R_A \vdash R_2 * R_B)$ is used to cancel, leaving the subgoal $R_A \vdash R_B$.

Then, if the remaining goal is emp $\vdash$ emp we are done; or if the remaining goal is $R_1 * \ldots * R_k \vdash ?U$, then U can be instantiated; otherwise the residual goal is left for the user.

In separation logic, it can happen that $R * R_1 \vdash R * R_2$ is provable where $R_1 \vdash R_2$ is not; so cancellation is not always the right tactic. But usually it is: it simplifies goals and often solves them entirely.

9.4 Magic wands and modalities

Our separation logic has the separating implication $-\!\!*$ and several modalities such as "later" $\triangleright$ and "in all worlds of the same age". These are rarely visible to users of VST-Floyd; mostly they're used in proving the soundness of our separation logic (Hoare rules). Some specialized proofs—such as defining a contravariant recursive type using the Löb fixpoint [3, Chapter 19]—do require the user to manipulate modalities and magic wand. The tactics most useful in those proofs are rewrite and sep_apply; we don't have special modality-handling tactics in our entailer.

On the other hand, there is some automatic modality handling in our forward tactic. Many of our Hoare logic rules have later $\triangleright$ modalities in their preconditions [3, pp. 96, 160–164]. For example, the rules SEMAX-SET, SEMAX-LOAD, SEMAX-STORE shown in §2.3 really have preconditions $\triangleright P$ (instead of P) and $\triangleright(P * p \mapsto \_)$ (instead of $P * p \mapsto \_$). We omitted the $\triangleright$ in §2.3 to simplify the presentation (though because $P \vdash \triangleright P$, the rules presented there are still sound). Now suppose the user's "current assertion" is $\mathsf{PROP}(\ldots)\mathsf{LOCAL}(\ldots)\mathsf{SEP}(A;\, \triangleright B;\, \triangleright C;\, D;\, E)$. Our forward tactic automatically applies rules such as $P \vdash \triangleright P$ and $\triangleright P * \triangleright Q \vdash \triangleright(P * Q)$ to prove that the current assertion entails $\triangleright\mathsf{PROP}(\ldots)\mathsf{LOCAL}(\ldots)\mathsf{SEP}(A;\, B;\, C;\, D;\, E)$; then the strong form of SEMAX-LOAD (etc.) can apply.

10 A worked example

Consider this C program:

```
int sumarray(int a[], int n) {
  int i,s,x;
  i=0;
  s=0;
  while (i<n) {
    x=a[i];
    s+=x;
    i++;
  }
  return s;
}
```

```
int four[4] = {1,2,3,4};

int main(void) {
  int s;
  s = sumarray(four,4);
  return s;
}
```

Verifiable C with VST-Floyd automation proves the correctness of this program quite straightforwardly, as we will show. Actually, this program is so simple, and its specification is so first-order, that weaker (but more automatic) systems such as Frama-C and VeriFast can prove its correctness as well. But the programs that illustrate VST's higher-order features would be too big to present here.

A reasonable *function specification* for `sumarray` is,

```
Definition sumarray_spec :=
DECLARE _sumarray
 WITH a: val, sh : share, σ : list Z, n: Z
 PRE [ _a OF (tptr tint), _n OF tint ]
    PROP (readable_share sh; 0 ≤ n ≤ Int.max_signed;
          Forall (fun x ⇒ Int.min_signed ≤ x ≤ Int.max_signed) σ)
```

LOCAL (temp _a a; temp _n (Vint (Int.repr n)))
SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a)
POST [tint]
PROP () LOCAL(temp ret_temp (Vint (Int.repr (fold_right Z.add 0 σ))))
SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a).

The names starting with underscore are C-language identifiers, and the italicized variables are Coq variables bound by WITH (which we wrote as Π in earlier sections).

The PROP part of the precondition says, there is some permission-share sh that grants at least permission to read from memory; there is a nonnegative (mathematical) integer n that is representable as a 32-bit signed integer; there is a sequence σ of (mathematical) integers, all of which in the range between the minimum and maximum signed 32-bit integers. The LOCAL part says, the C variable _a contains the value a, and _n contains the 32-bit representation of n. The SEP part says at address a there is an array[n] of C-language integers, and the contents of this array is the C-integer-value translation of the sequence σ.

The PROP part of the postcondition is empty, as usual, because every proposition in the precondition is still (necessarily) true. The LOCAL part of the postcondition describes the return value, the C pseudovariable called ret_temp, containing (the 32-bit representation of) the sum of σ. The SEP part says that this function has not altered the data structure in its input.

Now we prove of correctness of the program. We construct (the global part of) Δ from all the function specifications and global-variable types in the program and the standard library:

Definition Gprog : funspecs := ltac:(**with**_library prog [sumarray_spec; main_spec]).
Definition Vprog : varspecs. mk_varspecs prog. **Defined**.

Now we must prove that each function satisfies its specification. The form of the theorem is,

Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.
Proof.
start_function.

The semax_body judgment says, in the global context Vprog+Gprog, the syntactic function-definition f_summary (parsed by the front-end of CompCert) satisfies the Hoare specification sumarray_spec. To prove this, we use the tactic start_function. This constructs Δ from Vprog, Gprog, and the local-variable declarations of f_sumarray; it introduces the variables bound by the WITH clause of sumarray_spec, and does other bookkeeping.

Floyd introduces *MORE_COMMANDS* to stand for the "rest of the block," and we *abbreviate* this local definition to avoid clutter. Our abbreviate operator is just a way of hiding terms, taking advantage of Coq's "implicit argument" feature. Its definition uses braces to indicate that arguments A and x are implicit, and should not be displayed.

Definition abbreviate {A:Type} {x:A} := x.

Now our proof goal is,

```
a : val        sh : share
σ : list Z     n : Z
Δ := abbreviate : tycontext
SH : readable_share sh
H : 0 ≤ n ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ Int.min_signed ≤ x ≤ Int.max_signed) σ
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
----------------------------------------(1/1)
semax Δ
  (PROP ( )
   LOCAL (temp _a a; temp _n (Vint (Int.repr n)))
   SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a))
  (Ssequence (Sset _i (Econst_int (Int.repr 0) tint)) MORE_COMMANDS)
  POSTCONDITION
```

The proof goal is a Hoare triple for the entire function body, and the C command (written here in AST constructors) is `i=0; ...`. Two invocations of forward take us through the two assignment statements `i=0; s=0` to the proof state (we omit variable declarations $a, sh, \Delta, etc.$ above the line),

```
SH : readable_share sh
H : 0 ≤ n ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ Int.min_signed ≤ x ≤ Int.max_signed) σ
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
LOOP_BODY := abbreviate : statement
----------------------------------------(1/1)
semax Δ
  (PROP ( )
   LOCAL (temp _s (Vint (Int.repr 0)); temp _i (Vint (Int.repr 0));
   temp _a a; temp _n (Vint (Int.repr n)))
   SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a))
  (Ssequence (Swhile (Ebinop Olt (Etempvar _i tint) (Etempvar _n tint) tint)
                     LOOP_BODY) MORE_COMMANDS) POSTCONDITION
```

At the `while` loop, we apply forward_while, supplying a loop invariant in existentially quantified canonical form.

```
forward_while
 (EX i: Z, PROP (0 ≤ i ≤ n)
          LOCAL (temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr n));
                 temp _s (Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))))
          SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a)).
```

This leaves four subgoals. The first is to prove that the current precondition entails the loop invariant. This solves by simply Exists 0; entailer!

The second subgoal is to prove that the loop-test expression evaluates without getting "stuck" (i.e., doesn't refer to uninitialized variables, divide by zero, overflow, etc.). In this case, our computational typechecker [3, Chapter 25] calculates that the proof goal is True, because it has kept track that all the variables in `i<n` are initialized.

The third subgoal is to prove that the loop body preserves the loop invariant.

```
SH : readable_share sh
H : 0 ≤ n ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ Int.min_signed ≤ x ≤ Int.max_signed) σ
HRE : i < n
H1 : 0 ≤ i ≤ n
----------------------------------------(1/1)
semax Δ
 (PROP ( )
  LOCAL (temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr n));
         temp _s (Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))))
  SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a))
 (Ssequence
  (Sset _x (Ederef (Ebinop Oadd (Etempvar _a (tptr tint)) (Etempvar _i tint) (tptr tint)) tint))
  MORE_COMMANDS)
 POSTCONDITION
```

Before we go forward through `x=a[i];` it's helpful to assert that $|\sigma| = n$; this allows the LOAD rule to discharge one of its hypotheses. So we might try to write, assert(Zlength σ = n). But this is not provable from what's above the line. Instead, we need to use information from the data_at predicate in the precondition: that the contents of an array must be the same length as the array. In effect, we do this using the rule of consequence, to prove that the current precondition implies one that also has a PROP clause with $|\sigma| = n$. This is automated by the tactic,

```
assert_PROP (Zlength σ = n).
```

which leaves the subgoal,

```
ENTAIL Δ,
PROP ( )
LOCAL (temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr n));
       temp _s (Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))))
SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a)
⊢ !! (Zlength σ = n)
```

This is easily proved by (entailer!; rewrite !Zlength_map; re exivity).

Now, three invocations of forward take us to the end of the loop body, where we must prove that the current assertion entails the loop invariant:

```
H : 0 ≤ n ≤ Int.max_signed
HRE : i < n
H1 : 0 ≤ i ≤ n
----------------------------------------(1/1)
ENTAIL Δ,
PROP ( )
LOCAL (temp _i (Vint (Int.add (Int.repr i) (Int.repr 1)));
       temp _s (Vint (Int.add (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))
                              (Int.repr (Znth i σ 0))));
       temp _x (Vint (Int.repr (Znth i σ 0))); temp _a a; temp _n (Vint (Int.repr n)))
SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a)
⊢ EX j : Z,
    PROP (0 ≤ j ≤ n)
    LOCAL (temp _a a; temp _i (Vint (Int.repr j)); temp _n (Vint (Int.repr n));
           temp _s (Vint (Int.repr (fold_right Z.add 0 (sublist 0 j σ)))))
    SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a)
```

We instantiate the existential (j) with $i+1$, and invoke entailer!, leaving a residual proof goal of:

```
H : 0 ≤ Zlength σ ≤ Int.max_signed
H1 : 0 ≤ i ≤ Zlength σ
HRE : i < Zlength σ
----------------------------------------(1/1)
Vint (Int.repr (fold_right Z.add 0 (sublist 0 (i + 1) σ))) =
Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ) + Znth i σ 0))
```

This is proved by manipulations in the theory of sublists:

```
f_equal. f_equal. rewrite (sublist_split 0 i (i+1)) by omega.
rewrite sum_Z_app. rewrite (sublist_one i) with (d:=0) by omega.
simpl. rewrite Z.add_0_r. re  exivity.
```

The fourth subgoal (of forward_while) is to prove the remainder of the function-body after the while loop. What remains is just the `return` statement:

```
H : 0 ≤ n ≤ Int.max_signed
H0 : Forall (fun x : Z ⇒ Int.min_signed ≤ x ≤ Int.max_signed) σ
HRE : i ≥ n
H1 : 0 ≤ i ≤ n
POSTCONDITION := abbreviate : ret_assert
----------------------------------------(1/1)
semax Δ
  (PROP ( )
   LOCAL (temp _a a; temp _i (Vint (Int.repr i)); temp _n (Vint (Int.repr n));
          temp _s (Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))))
   SEP (data_at sh (tarray tint n) (map Vint (map Int.repr σ)) a))
  (Sreturn (Some (Etempvar _s tint)))
  POSTCONDITION
```

Invoking forward through the `return s;` yields an entailment goal: the current assertion implies the function's postcondition. The entailer! leaves this residual goal:

H1 : $0 \leq i \leq$ Zlength (map Vint (map Int.repr σ))
HRE : $i \geq$ Zlength (map Vint (map Int.repr σ))
----------(1/1)
Vint (Int.repr (fold_right Z.add 0 σ)) = Vint (Int.repr (fold_right Z.add 0 (sublist 0 i σ)))

This is easily proved by the theory of sublists:

autorewrite **with** sublist **in** $\star$. autorewrite **with** sublist. re exivity.

This case study can be found in our Coq development: progs/verif_sumarray.v.

11 Use cases

Verifiable C with VST-Floyd has been used to prove several small, but real, C programs correct with respect to functional specifications:

	Queue	SHA	HMAC	HKDF	DRBG	Sort	Salsa	AES	Mailbox
Lines of C[a]	82	239	256	100	380	40	280	240	180
Lines of Floyd[a]	530	4956[c]	5432	796	5919	573	5019	1256	2860
Proof-check time[e]	84	1120	980	670	4100	253	1730		
Years of Coq experience before beginning the proof	6	7	4	6	0.3	4	5	2	7
Years of VST experience	1	2	0	2	0	0	1	0.02	0.4
Weeks of effort[f]	2[dg]	20[dg]	5	4	3		6	25[d]	10[d]

[a]Line counts include blank lines and comments. [b]Many of these lines are quite long. [c]Plus 1598 lines of Coq proofs about properties of the functional specification. [d]Includes work to build or improve Floyd itself. [e]Timings measured in seconds on Intel Core i7 (at 3.7 Ghz), one processor, with plenty of cache and 32GB RAM, but no proof requires more than 2GB RAM (and in practice, Coq parallelizes well by `make -j` and other means). [f]Very rough approximation. [g]Done in an early, primitive version of VST-Floyd, though the line-counts given are in current VST-Floyd.

Queue. Linked-list imperative FIFO queue abstract-data-type, proved by Andrew Appel, 2012. [3, Chapter 28]

SHA-256. The OpenSSL implementation of the SHA-2 cryptographic hash algorithm, specialized to the 256-bit case. Proof done by Andrew Appel, 2013. [2]

HMAC. The OpenSSL implementation of the HMAC cryptographic authentication algorithm; proof done by Lennart Beringer, 2014. [7] In this case and the DRBG case, the proof that the C program implements its functional spec is done by the author we name here; the cited papers also include other important proofs, e.g., that the functional spec has the appropriate cryptographic properties.

HKDF. BoringSSL's implementation of HMAC-based key derivation function; proof done by Lennart Beringer, 2017.

DRBG. The mbedTLS implementation of the HMAC-DRBG random-number generator; proof done by Naphat Sanguansin, 2015.

Sort. In-place merge sort of linked lists, proved by Jean-Marie Madiot, 2015.

Salsa. Parts of TweetNaCl's implementation of the stream cipher Salsa20 [8]; partially done by Lennart Beringer, 2016.

AES. The mbedTLS implementation of AES-256 symmetric-key encryption; proof done by Samuel Gruetter, 2017.

Mailbox. A novel concurrent-shared-memory communications protocol, proof done in Concurrent Verifiable C (with VST-Floyd) by William Mansky, 2016. [26]

12 Discussion and Related Work

In the development of VST-Floyd, we made some design decisions differently from other verification tools. In this section, we discuss these choices.

12.1 Automatic vs. interactive tools

Automatic verification tools such as Dafny [24], Frama-C [30], Hip/Sleek [10], and CBMC [12] achieve great success in their application domain. But in the domain of functional-correctness verification with higher-order logic predicates, fully automatic decision procedures cannot be effective—some interaction is necessary. VST-Floyd is an interactive tool, to support application-domain reasoning (in an expressive dependently typed higher-order logic) that is beyond the scope of a general tool (and may itself be undecidable).

Automatic tools put restrictions on their application domains so that they will not have this undecidability problem. Some tools do only shape analysis instead of functional correctness. In some tools, assertions must be first-order and the predicates in assertions are limited.

To enable VST-Floyd to be a general purpose tool for functional verification, we build it in Coq, a general purpose proof assistant in which domain-specific definitions and proofs can be formalized—our system allows users to apply arbitrary domain-specific theories.

12.2 Verification condition generation

Formalizing program logics in proof assistants and formally proving their soundness has been an active area of research for many years. However, combining mechanized proofs of metatheoretical aspects—formal soundness and (relative) completeness of the logic with respect to an operational semantics, as for example carried out by Kleymann [20] and Nipkow [28]—with efficient verification engines has long been an elusive goal. We are unaware of prior work that also includes a provably sound connection to a verified compiler.

Wildmoser [31] presented formal proofs of soundness and completeness of a VCG framework in Isabelle/HOL that is parametric in the safety logic, programming language, and concrete safety policy, and specializes this framework to typical proof-carrying-code-style safety policies and the JINJA subset of Java bytecode. Matthews *et al.* directly derive verification conditions in the proof assistant ACL2 from the operational semantics, obviating the need to formulate a separate VCG framework [27]. Neither of these works exploit the structuring mechanisms of separation logics; both avoid the intricacies of C, particularly concerning the memory model.

12.3 Canonical forms of separation logic assertions

Berdine *et al.* [6] first proposed a canonical form of separation logic which distinguishes pure facts and spatial facts. Specifically, an assertion of their canonical form can be represented as $(P_1 \wedge \cdots \wedge P_n) \wedge (Q_1 * \ldots * Q_m)$ in which P_i are pure facts and Q_i are spatial facts. They did not isolate program variables as we do.

Charge! [5] is a separation logic tool proved sound in Coq. Like ours, their assertions are predicates over stack-heap pairs. In comparison, the separating conjuncts in their "canonical form" of assertions are not required to be independent of program variables. As a consequence, generating a strongest postcondition is more complicated.

Iris Proof Mode (discussed below) has an assertion form $\square P * L \mathrel{-\!\!*} R$ where P are the persistent conjuncts, L and R are separating conjunctions.

12.4 Systems for machine-checked separation-logic program proofs

Charge! [5], Bedrock [11], and Iris [19] have very similar design philosophies to VST-Floyd. All of them are shallowly embedded separation logics supporting interactive proofs in Coq of the functional correctness of pointer-manipulating programs.

Bedrock is a program logic and tool for reasoning about low-level (idealized assembly language) programs. Its assertion language uses lambda calculus and directly refers to stack and heap. To express the following canonical assertion in VST-Floyd,

$$\mathsf{PROP}()\mathsf{LOCAL}(\mathsf{temp\ x}\ p)\mathsf{SEP}(p \mapsto 0)$$

Bedrock users directly write: $\lambda s : \mathsf{stack}.\ \lambda h : \mathsf{heap}.\ s(\mathsf{x}) = p \wedge h(p) = 0$.

Floyd performs better for two reasons. On one hand, the abstraction of separating conjunction enables concise representation of heap disjointness. $\mathsf{SEP}(p \mapsto 0; q \mapsto 0)$ implies the fact the $p \neq q$. On the other hand, Coq's tactic language does not work well in a subcontext. In the example above, the expression $s(\mathsf{x}) = p \wedge h(p) = 0$ is in a subcontext with two extra variables s and h. Pattern matching in Coq's tactic language does not work very well in this scenario. Our separation logic predicates abstract away this lambda calculus. Our tactic library directly operates in the top level context and is thus more efficient.

Charge! is a program logic for Java, based on separation logic. It is not linked to a formally verified Java compiler, which means that one cannot consider its Java

semantics fully "debugged." Charge's assertion language is mostly written in lifted separation logic while our canonical form mostly operates in the unlifted language (SEP part). Because of the isolation of C variables in our canonical form, our tactic library only modifies one conjunct (in LOCAL part or SEP part) in a precondition to generate a postcondition. Until 2014, assertions in VST-Floyd were written in the lifted language instead of canonical forms. We found that proof rules were very complicated and tactics were very slow. For similar reasons, Charge! is less efficient than VST-Floyd.

Iris [19] is a general purpose modal concurrent separation logic, parameterized over programming languages and program semantics, embedded in (and proved sound in) Coq. Iris Proof Mode (IPM) [23] provides tactics and lemmas for separation-logic proofs in Iris.

IPM has several features lacking in VST-Floyd: the ability to name the left-hand-side conjuncts as individual (separating) hypotheses, special handling of a "persistent" modality, and special tactics for the various modalities of the Iris logic [22]. Some of these modalities are also in VST's logic (see §9.4), but without special support from Floyd. Particularly interesting in IPM is support for deriving Hoare rules by proofs at the logic level, where VST has such proofs at the model level; IPM's modalities make this possible. Because VST's underlying logic and semantic model can support these modalities, it would be a worthwhile improvement to import this idea from IPM into Floyd.

Unlike Floyd, as of 2017 IPM has no reported support for aggregate types (as in §3 of this paper) or efficient treatment of program variables (as in §5). IPM can do forward symbolic execution, but it is less automated than Floyd: the user must apply more per-statement tactics, and the handling of function-call postconditions, if-statements, and while-loops (or the recursive-function equivalent) is less automated. Because IPM is designed for ML-like languages whose local variables are substitution-based, IPM naturally lacks automation for Algol-style (C-style) local variables (as in §5,§9.3).

As Krebbers *et al.* write, "[unlike IPM] all the tools that we are aware of are primarily focused on program verification." Indeed, that is the focus of VST-Floyd. However, it should be possible to include some of our techniques into IPM and get the best of both worlds.

CakeML [15] is now accompanied with a separation Hoare logic for verifying functional correctness. Their system is based on a Characteristic Formulae framework, which is similar to Hoare logic. The construction of Characteristic Formulae is actually a combination of Hoare rules and the definition of Hoare triple validity. The soundness of such construction corresponds to the soundness of Hoare rules.

12.5 Dependent types used in separation logic predicates

Affeldt and Marti [1] and Krebbers [21] also have separation-logic "maps-to" operators $p \mapsto_\tau v$ parameterized by a type τ that may be an aggregate type such as struct. However, where our reptype(τ) operator, where for example $\tau =$(struct foo {int x,y;}), calculates the type of v to be simply val $\times$ val, in those systems their reptype (called

log by Affeldt) is an inductively defined type. The advantage of their approach is a simpler construction of the metatheory (no need for an explicit ranking to ensure C types are acyclic, less manipulation of dependent types). The disadvantage is that—although in principle there's a type isomorphism between $\mathsf{log}(\tau)$ and $\mathsf{val} \times \mathsf{val}$—the user cannot simply apply the familiar operators fst,snd and constructors $(\_, \_)$ for Cartesian product, but must deconstruct the log operator by case analysis.

Our data_at predicate, applied to a C array type, relates the contents of the array to a sequence. The length of the sequence must, of course, be the same as the length of the array. How should this be represented in Coq? We faced the design choice: list, with a separate proposition about the length of the list; or dependently typed "vector," which includes the length in the Coq type of the sequence. The two approaches are equally powerful. We chose the nondependently typed lists, because we feel that the proof theory is simpler and more tractable for users. Our entailer's *saturate local* phase (§9.3) automatically puts the list-length facts above the line for the user.

13 Conclusion

We have presented VST-Floyd, an extensive collection of proof tactics, abstraction principles and other automation features that turn Verifiable C into a practically useful verification tool for nontrivial C programs. While the present description focused on the verification of sequential code, VST's semantic model also supports reasoning about multithreaded code, as demonstrated by a recent case study on a mailbox communication protocol with fine-grain concurrency primitives [26]. Specifying the invariants of thread primitives implicitly requires reasoning about function pointers, for which we are developing additional automation support at present. Further ongoing work includes the support of 64-bit architectures as recently enabled by CompCert3.0, and the extension of VST's soundness guarantee w.r.t. processor models that exhibit relaxed cache coherence.

In summary, VST-Floyd represents a key component of the CompCert/VST infrastructure. For those system components that require high assurance (necessitating the use of formal verification) and also high performance (suggesting the use of C) the Verified Software Toolchain enables practical, end-to-end modular program verification.

References

1. Reynald Affeldt and Nicolas Marti. Towards formal verification of TLS network packet processing written in C. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification*, pages 35–46. ACM, 2013.
2. Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015.
3. Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.
4. Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, September 2001.

5. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - a framework for higher-order separation logic in Coq. In *ITP*, pages 315–331, 2012.
6. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.
7. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221. USENIX Association, August 2015.
8. Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer, 2014.
9. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44:289–300, January 2009.
10. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
11. Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP'13: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, September 2013.
12. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
13. Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposium in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967.
14. Samuel Gruetter. Improving the Coq proof automation tactics of the Verified Software Toolchain, based on a case study on verifying a C implementation of the AES encryption algorithm. Master thesis. Ecole Polytechnique Fédérale de Lausanne, 2017.
15. Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. 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*, pages 584–610, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
16. C A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):578–580, October 1969.
17. Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proc. 37th Annual ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 171–185, January 2010.
18. Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 523–536. ACM, 2013.
19. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.
20. Thomas Kleymann. *Hoare logic and VDM : machine-checked soundness and completeness proofs*. PhD thesis, University of Edinburgh, UK, 1998.
21. Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University, December 2015.
22. Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.

23. Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017.
24. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
25. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
26. William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. In *Proceedings of the 2017 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '17. ACM, 2017.
27. John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2006.
28. Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In Julian C. Bradfield, editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2002.
29. John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.
30. Julien Signoles. Foncteurs impératifs et composés: la notion de projets dans Frama-C. In Alan Schmitt, editor, *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, 2009.
31. Martin Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Technical University Munich, November 2005.