Chapter 1

Introduction

An exciting development of the 21st century is that the 20th-century vision of mechanized program verification is finally becoming practical, thanks to 30 years of advances in logic, programming-language theory, proof-assistant software, decision procedures for theorem proving, and even Moore's law which gives us everyday computers powerful enough to run all this software.

We can write functional programs in ML-like languages and prove them correct in expressive higher-order logics; and we can write imperative programs in C-like languages and prove them correct in appropriately chosen program logics. We can even prove the correctness of the verification toolchain itself: the compiler, the program logic, automatic static analyzers, concurrency primitives (and their interaction with the compiler). There will be few places for bugs (or security vulnerabilities) to hide.

This book explains how to construct powerful and expressive program logics based on separation logic and Indirection Theory. It is accompanied by an open-source machine-checked formal model and soundness proof, the *Verified Software Toolchain*[1] *(VST)*, formalized in the Coq proof assistant. The VST components include the theory of *separation logic* for reasoning about pointer-manipulating programs; *indirection theory* for reasoning with "step-indexing" about first-class function pointers, recursive types,

[1]`http://vst.cs.princeton.edu`

recursive functions, dynamic mutual-exclusion locks, and other higher-order programming; a *Hoare logic* (separation logic) with full reasoning about control-flow and data-flow of the C programming language; theories of *concurrency* for reasoning about programming models such as Pthreads; theories of *compiler correctness* for connecting to the CompCert verified C compiler; theories of *symbolic execution* for implementing foundationally verified static analyses. VST is built in a modular way, so that major components apply very generally to many kinds of separation logics, Hoare logics, and step-indexing semantics.

One of the major demonstration applications comprises certified program logics and certified static analyses for the *C light* programming language. C light is compiled into assembly language by the CompCert[2] certified optimizing compiler. [62] Thus, the VST is useful for verified formal reasoning about programs that will be compiled by a verified compiler. But Parts I, II, and V of this book show principles and Coq developments that are quite independent of CompCert and have already been useful in other applications of separation logics.

PROGRAM LOGICS FOR CERTIFIED COMPILERS. Software is complex and prone to bugs. We would like to reason about the correctness of programs, and even to prove that the behavior of a program adheres to a formal specification. For this we use program logics: rules for reasoning about the behavior of programs. But programs are large and the reasoning rules are complex; what if there is a bug in our proof (in our application of the rules of the program logic)? And how do we know that the program logic itself is sound—that when we conclude something using these rules, the program will really behave as we concluded? And once we have reasoned about a program, we compile it to machine code; what if there is a bug in the compiler?

We achieve soundness by formally verifying our program logics, static analyzers, and compilers. We prove soundness theorems based on foundational specifications of the underlying hardware. We check all proofs by machine, and connect the proofs together end-to-end so there are no gaps.

[2] `http://compcert.inria.fr`

DEFINITIONS. A *program* consists of instructions written in a *programming language* that direct a computer to perform a task. The *behavior* of a program, *i. e.* what happens when it executes, is specified by the *operational semantics* of the programming language. Some programming languages are *machine languages* that can directly execute on a computer; others are *source languages* that require translation by a *compiler* before they can execute.

A *program logic* is a set of formal rules for *static* reasoning about the behavior of a program; the word *static* implies that we do not actually execute the program in such reasoning. *Hoare logic* is an early and still very important program logic. *Separation logic* is a 21st-century variant of Hoare logic that better accounts for pointer and array data structures.

A compiler is *correct* with respect to the specification of the operational semantics of its source and its target languages if, whenever a source program has a particular defined behavior, and when the compiler translates that program, then the target program has a *corresponding* behavior. [38] The correspondence is part of the correctness specification of the compiler, along with the two operational semantics. A compiler is *proved correct* if there is a formal proof that it meets this specification. Since the compiler is itself a program, this formal proof will typically be using the rules of a program logic for the implementation language of the compiler.

Proofs in a logic (or program logic) can be written as derivation trees in which each node is the application of a rule of the system. The validity of a proof can be checked using a computer program. A *machine-checked proof* is one that has been checked in this way. Proof-checking programs can be quite small and simple, [12] so one can reasonably hope to implement a proof-checker free of bugs.

It is inconvenient to construct derivation trees "by hand." A *proof assistant* is a tool that combines a proof checker with a user interface that assists the human in building proofs. The proof assistant may also contain algorithms for proof automation, such as *tactics* and *decision procedures.*

A *certified compiler* is one proved correct with a machine-checked proof. A *certified program logic* is one proved sound with a machine-checked proof. A *certified program* is one proved correct (using a program logic) with a machine-checked proof.

A *static analysis* algorithm calculates properties of the behavior of a program without actually running it. A static analysis is *sound* if, whenever it claims some property of a program, that property holds on all possible behaviors (in the operational semantics). The proof of soundness can be done using a (sound) program logic, or it can be done directly with respect to the operational semantics of the programming language. A *certified static analysis* is one that is proved sound with a machine-checked proof—either the static analysis program is proved correct, or each run of the static analysis generates a machine-checkable proof about a particular instance.

In Part I we will review Hoare logics, operational semantics, and separation logics. For a more comprehensive introduction to Hoare logic, the reader can consult Huth and Ryan [54] or many other books; For operational semantics, see Harper [47, Parts I & II] or Pierce [75]. For an introduction to theorem-proving in Coq, see Pierce's *Software Foundations*[76] which also covers applications to operational semantics and Hoare logic.

THE VST SEPARATION LOGIC FOR C LIGHT is a higher-order impredicative concurrent separation logic certified with respect to CompCert. *Separation logic* means that its assertions specify heap-domain footprints: the assertion $(p \mapsto x) * (q \mapsto y)$ describes a memory with exactly two disjoint parts; one part has only the cell at address p with contents x, and the other has only address q with contents y, with $p \neq q$. *Concurrent* separation logic is an extension that can describe shared-memory concurrent programs with Dijkstra-Hoare synchronization (e.g., Pthreads). *Higher-order* means that assertions can use existential and universal quantifiers, the logic can describe pointers to functions and mutex locks, and recursive assertions can describe recursive data types such as lists and trees. *Impredicative* means that the $\exists$ and $\forall$ quantifiers can even range over assertions containing quantifiers. *Certified* means that there is a machine-checked proof of soundness with respect to the operational semantics of a source language of the CompCert C compiler.

A separation logic has assertions $p \mapsto x$ where p ranges over a particular address type A, x ranges over a specific type V of values, and the assertion as a whole can be thought of as a predicate over some specific type of "heaps" or "computer memories" M. Then the logic will have theorems such as $(p \mapsto x) * (q \mapsto y) \vdash (q \mapsto y) * (p \mapsto x)$.

We will write down *generic* separation logic as a theory parameterizable by types such as A, V, M, and containing generic axioms such as $P * Q \vdash Q * P$. For a particular instantiation such as CompCert C light, we will instantiate the generic logic with the types of C values and C expressions.

Chapter 3 will give an example of an informal program verification in "pencil-and-paper" separation logic. Then Part V shows the VST tools applied to build a foundationally sound toolchain for a toy language, with a machine-verified separation-logic proof of a similar program. Part III demonstrates the VST tools applied to the C language, connected to the CompCert compiler, and shows machine-checked verification C programs.

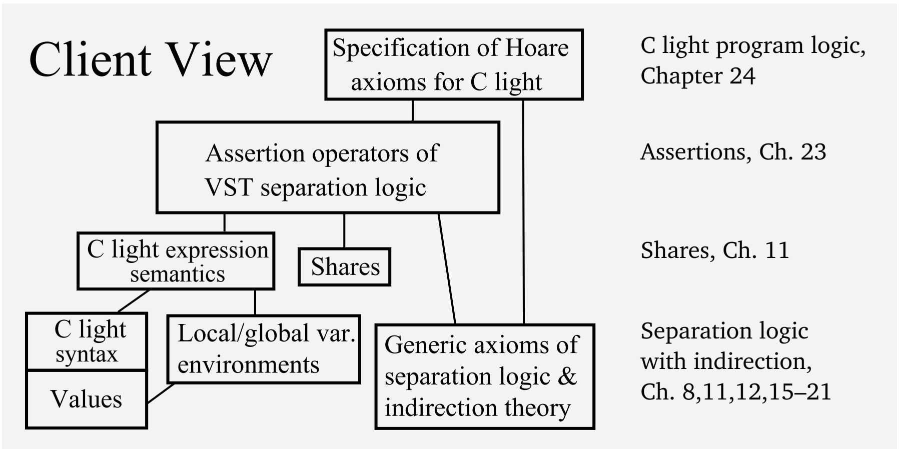
Client View
Specification of Hoare axioms for C light
C light program logic, Chapter 24
Assertion operators of VST separation logic
Assertions, Ch. 23
C light expression semantics
Shares
Shares, Ch. 11
C light syntax
Values
Local/global var. environments
Generic axioms of separation logic & indirection theory
Separation logic with indirection, Ch. 8,11,12,15–21

Figure 1.1: Client view of VST separation logic

FIGURE 1.1 SHOWS THE *client view* of the VST separation logic for *C light*—that is, the specification of the axiomatic semantics. Users of the program logic will reason directly about CompCert values (integers, floats, pointers) and C-light expression evaluation. Users do not see the operational semantics of C-light commands, or CompCert memories. Instead, they use

the axiomatic semantics—the Hoare judgment and its reasoning rules—to reason indirectly about memories via assertions such as $p \mapsto x$.

The modular structure of the *client view* starts (at bottom left of Fig. 1.1) with the specification of the C light language, a subset of C chosen for its compatibility with program-verification methods. We have *C* values (such as integers, floats, and pointers); the abstract syntax of C light, and the mechanism of evaluating C light expressions. The client view treats statements such as assignment and looping *abstractly* via an axiomatic semantics (Hoare logic), so it does not expose an operational semantics.

At bottom right of Figure 1.1 we have the operators and axioms of separation logic and of indirection theory. At center are the assertions of our program logic for C light, which (as the diagram shows) make use of C-light expressions and of our logical operators. At top, the Hoare axioms for C light complete the specification of the program logic.

Readers primarily interested in *using* the VST tools may want to read Parts I through III, which explain the components of the client view.

THE SOUNDNESS PROOF OF THE VST SEPARATION LOGIC is constructed by reasoning in the *model* of separation logic. Figure 1.2 shows the structure of the soundness proof. At bottom left is the specification of C-light operational semantics. We have a generic theory of safety and simulation for shared-memory programs, and we instantiate that into the "C light safety" theory.

At bottom right (Fig. 1.2) is the theory of *separation algebras*, which form models of separation logics. The assertions of our logic are predicates on the *resource maps* that, in turn, model CompCert memories. The word *predicate* is a technical feature of our Indirection Theory that implicitly accounts for "resource approximation," thus allowing higher-order reasoning about circular structures of pointers and resource invariants.

We construct a semantic model of the Hoare judgment, and use this to prove sound all the judgment rules of the separation logic. All this is encapsulated in a Coq module called SeparationLogicSoundness.

Parts IV through VI explain the components of Figure 1.2, the semantic model and soundness proof of higher-order impredicative separation logic for CompCert C light.

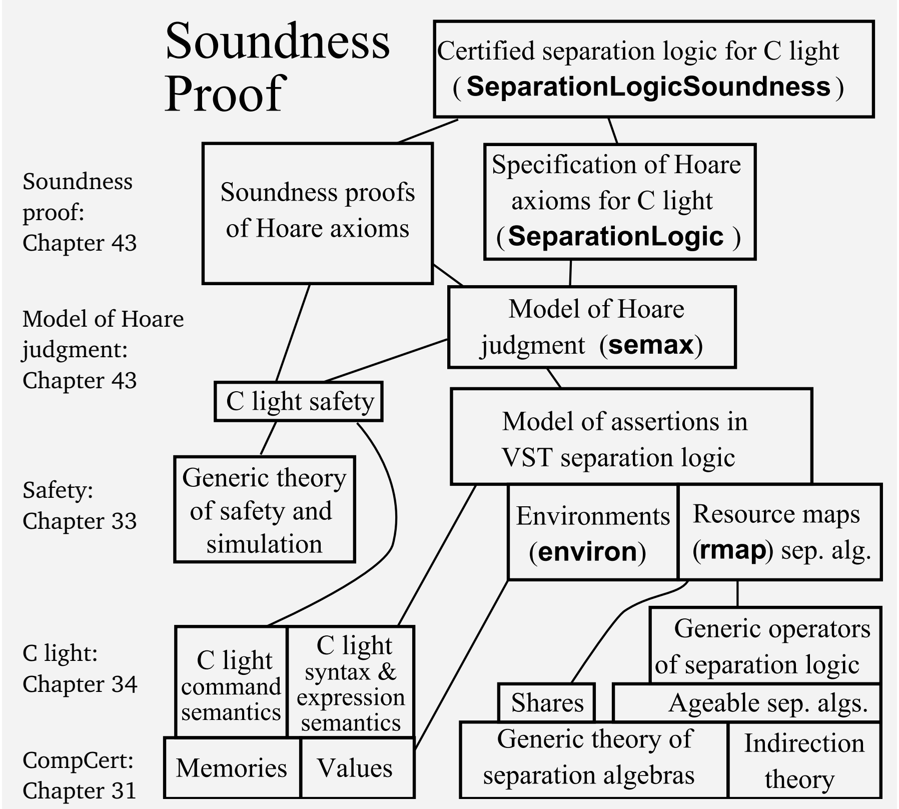
Soundness Proof
Certified separation logic for C light (**SeparationLogicSoundness**)
Soundness proof: Chapter 43
Soundness proofs of Hoare axioms
Specification of Hoare axioms for C light (**SeparationLogic**)
Model of Hoare judgment: Chapter 43
Model of Hoare judgment (**semax**)
C light safety
Model of assertions in VST separation logic
Safety: Chapter 33
Generic theory of safety and simulation
Environments (**environ**)
Resource maps (**rmap**) sep. alg.
Generic operators of separation logic
C light: Chapter 34
C light command semantics
C light syntax & expression semantics
Shares
Ageable sep. algs.
CompCert: Chapter 31
Memories
Values
Generic theory of separation algebras
Indirection theory

Figure 1.2: Structure of the separation-logic soundness proof

The Coq development of the Verified Software Toolchain is available at `vst.cs.princeton.edu` and is structured in a root directory with several subdirectories:

compcert: A few files copied from the CompCert verified C compiler, that comprise the specification of the C light programming language.

sepcomp: Theory of how to specify shared-memory interactions of CompCert-compiled programs.

msl: Mechanized Software Library, the theory of separation algebras, share accounting, and generic separation logics.

veric: The program logic: a higher-order splittable-shares concurrent separation logic for C light.

floyd: A proof-automation system of lemmas and tactics for semiautomated application of the program logic to C programs (named after Robert W. Floyd, a pioneer in program verification).

progs: Applications of the program logic to sample programs.

veristar: A heap theorem prover using resolution and paramodulation.

A proof development, like any software, is a living thing: it is continually being evolved, edited, maintained, and extended. We will not tightly couple this book to the development; we will just explain the key mathematical and organizational principles, illustrated with snapshots from the Coq code.

Part I

Generic separation logic

SYNOPSIS: *Separation logic is a formal system for static reasoning about pointer-manipulating programs. Like Hoare logic, it uses assertions that serve as preconditions and postconditions of commands and functions. Unlike Hoare logic, its assertions model anti-aliasing via the disjointness of memory heaplets. Separation algebras serve as models of separation logic. We can define a calculus of different kinds of separation algebras, and operators on separation algebras. Permission shares allow reasoning about shared ownership of memory and other resources. In a first-order separation logic we can have predicates to describe the contents of memory, anti-aliasing of pointers, and simple (covariant) forms of recursive predicates. A simple case study of straight-line programs serves to illustrate the application of separation logic.*

Chapter 2

Hoare logic

Hoare logic is an axiomatic system for reasoning about program behavior in a programming language. Its judgments have the form $\{P\}\,c\,\{Q\}$, called *Hoare triples*.[1] The command c is a statement of the programming language. The precondition P and postcondition Q are assertions characterizing the state before and after executing c.

In a Hoare logic of *total correctess*, $\{P\}\,c\,\{Q\}$ means, "starting from any state on which the assertion P holds, execution of the command c will safely terminate in a state on which the assertion Q holds."

In a Hoare logic of *partial correctness*, $\{P\}\,c\,\{Q\}$ means, "starting from any state on which the assertion P holds, execution of the command c will either infinite loop or safely terminate in a state on which the assertion Q holds." This book mainly addresses logics of partial correctness.[2]

[1]Hoare wrote his triples $P\{c\}Q$ with the braces quoting the commands, which makes sense *when quoting program commands within a logical statement.* Wirth used the braces as comment brackets in the Pascal language to encourage assertions as comments, leading to the style $\{P\}c\{Q\}$, which makes more sense *when quoting assertions within a program.* The Wirth style is now commonly used everywhere, regardless of where it makes sense.

[2] Some of our semantic techniques work best in a partial-correctness setting. We make the excuse that total correctness—knowing that a program terminates—is little comfort without also knowing that it terminates in less than the lifetime of the universe. It is better to have a *resource bound,* which is actually a form of partial correctness. Our techniques do extend to logics of resource-bounds [39].