Excerpt: Table of Contents and first three chapters

# PROGRAM LOGICS

## FOR CERTIFIED COMPILERS

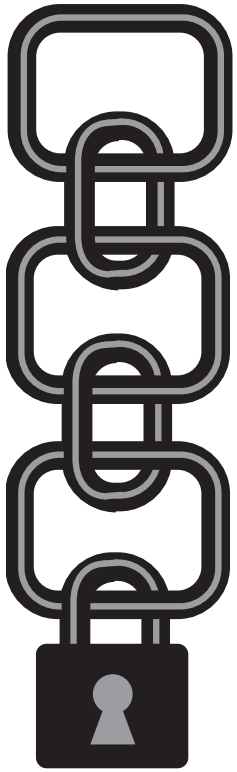
ANDREW W. APPEL

ROBERT DOCKINS, AQUINAS HOBOR, LENNART BERINGER, JOSIAH DODDS, GORDON STEWART, SANDRINE BLAZY, XAVIER LEROY

# PROGRAM LOGICS FOR CERTIFIED COMPILERS

Separation logic is the twenty-first-century variant of Hoare logic that permits verification of pointer-manipulating programs. This book covers practical and theoretical aspects of separation logic at a level accessible to beginning graduate students interested in software verification. On the practical side it offers an introduction to verification in Hoare and separation logics, simple case studies for toy languages, and the Verifiable C program logic for the C programming language. On the theoretical side it presents separation algebras as models of separation logics; step-indexed models of higher-order logical features for higher-order programs; indirection theory for constructing step-indexed separation algebras; tree-shares as models for shared ownership; and the semantic construction (and soundness proof) of Verifiable C. In addition, the book covers several aspects of the CompCert verified C compiler, and its connection to foundationally verified software analysis tools. All constructions and proofs are made rigorous and accessible in the Coq developments of the open-source Verified Software Toolchain.

Andrew W. Appel is the Eugene Higgins Professor and Chairman of the Department of Computer Science at Princeton University, where he has been on the faculty since 1986. His research is in software verification, computer security, programming languages and compilers, automated theorem proving, and technology policy. He is known for his work on Standard ML of New Jersey and on Foundational Proof-Carrying Code. He is a Fellow of the Association for Computing Machinery, recipient of the ACM SIGPLAN Distinguished Service Award, and has served as Editor-in-Chief of *ACM Transactions on Programming Languages and Systems*. His previous books include *Compiling with Continuations* (1992), the *Modern Compiler Implementation* series (1998 and 2002), and *Alan Turing's Systems of Logic* (2012).



# PROGRAM LOGICS FOR CERTIFIED COMPILERS

ANDREW W. APPEL
*Princeton University, Princeton, New Jersey*

ROBERT DOCKINS
*Portland State University, Portland, Oregon*

AQUINAS HOBOR
*National University of Singapore and Yale/NUS College, Singapore*

LENNART BERINGER
*Princeton University, Princeton, New Jersey*

JOSIAH DODDS
*Princeton University, Princeton, New Jersey*

GORDON STEWART
*Princeton University, Princeton, New Jersey*

SANDRINE BLAZY
*Université de Rennes 1*

XAVIER LEROY
*INRIA Paris-Rocquencourt*

CAMBRIDGE
UNIVERSITY PRESS
32 Avenue of the Americas, New York, NY 10013-2473, USA

Cambridge University Press is part of the University of Cambridge

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107048010

© Andrew W. Appel 2014

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2014

Printed in the United States of America

*A catalog record for this publication is available from the British Library.*

ISBN 978-1-107-04801-0 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Web sites referred to in this publication and does not guarantee that any content on such Web sites is, or will remain, accurate or appropriate.

This book is typeset in the Bitstream Charter font.
Font Copyright ©1989–1992, Bitstream Inc., Cambridge, MA.

*in memory of*

*Kenneth I. Appel*
*1932–2013*

*a pioneer in computer proof*

# *Contents*

## *Road map*

Readers interested in **the theory of separation logic** (with some example applications) should read Chapters 1–21. Readers interested in **the use of separation logic to verify C programs** should read Chapters 1–6 and 8–30. Those interested in **the theory of step-indexing** and **indirection theory** should read Chapters 35–39. Those interested in building models of **program logics** proved sound **for certified compilers** should read Chapters 40–47, though it would be helpful to read Chapters 1–39 as a warm-up.

# *Acknowledgments*

I thank Jean-Jacques Lévy for hosting my visit to INRIA Rocquencourt 2005–06, during which time I started thinking about the research described in this book. I enjoyed research collaborations during that time with Francesco Zappa Nardelli, Sandrine Blazy, Paul-André Melliès, and Jérôme Vouillon.

I thank the scientific team that built and maintains the Coq proof assistant, and I thank INRIA and the research funding establishment of France for supporting the development of Coq over more than two decades.

Mario Alvarez and Margo Flynn provided useful feedback on the usability of VST 0.9.

Research funding for some of the scientific results described in this book was provided by the Air Force Office of Scientific Research (agreement FA9550-09-1-0138), the National Science Foundation (grant CNS-0910448), and the Defense Advanced Research Projects Agency (agreement FA8750-12-2-0293). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR, NSF, DARPA, or the U.S. government.

# *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.

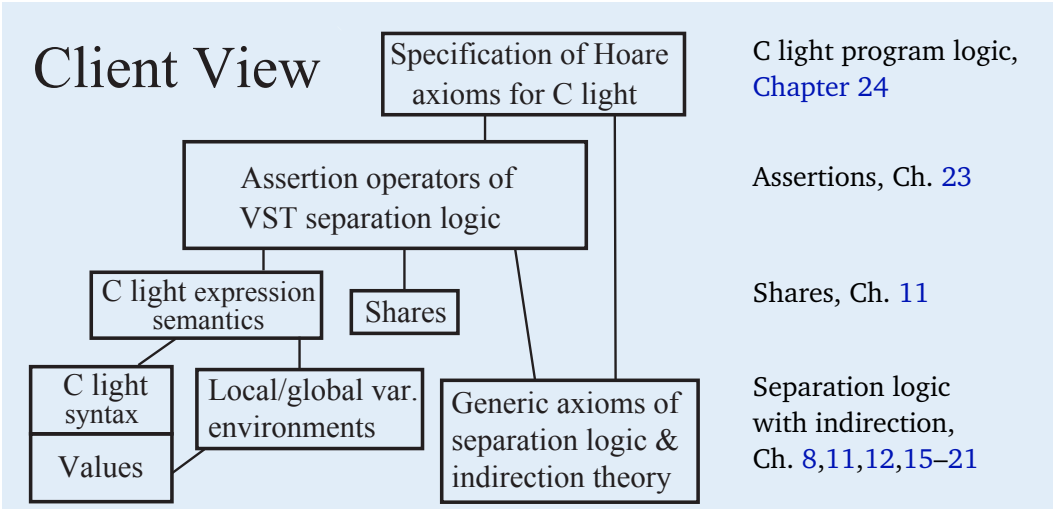



Figure 1.1: Client view of VST separation logic

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.

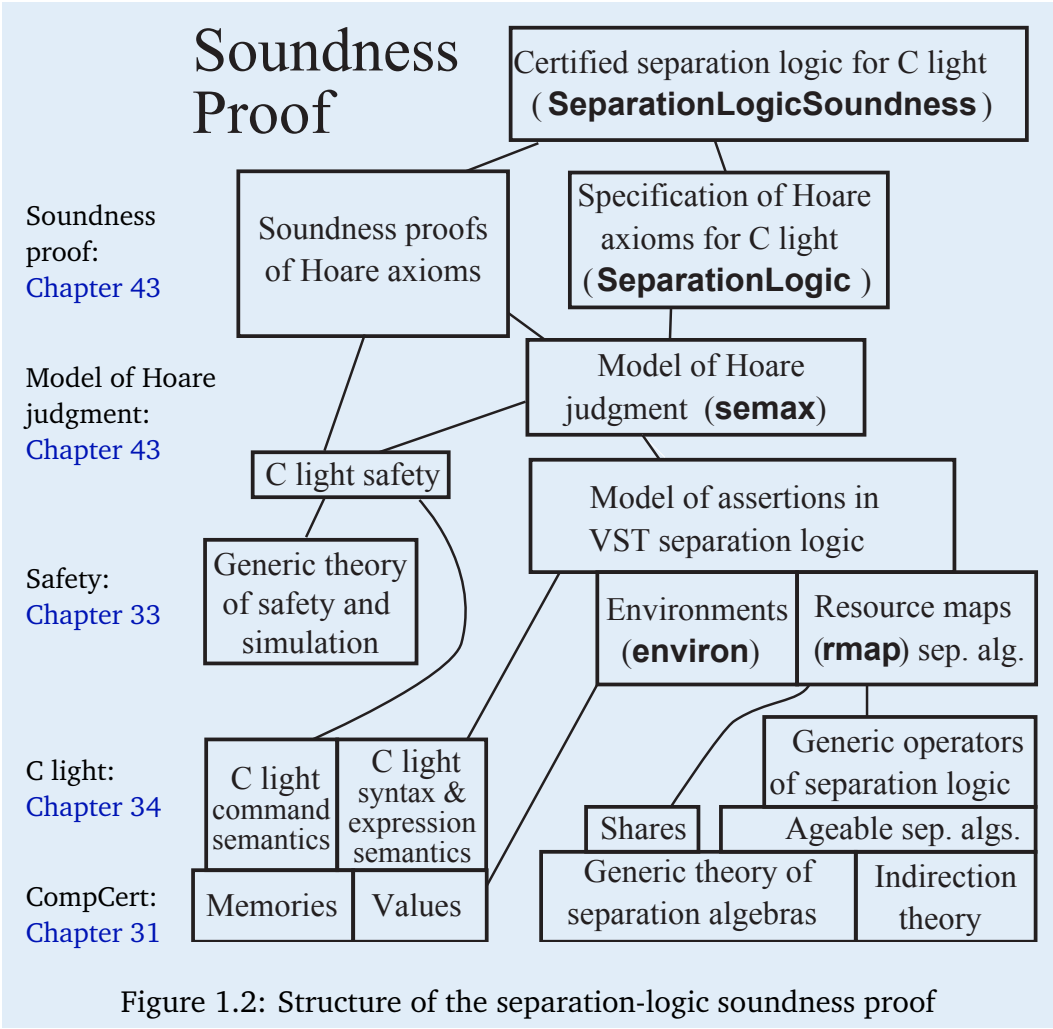



Figure 1.2: Structure of the separation-logic soundness proof

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].

THE INFERENCE RULES OF HOARE LOGIC include,

$$\text{seq}\ \frac{\{P\}\,c_1\,\{P'\} \qquad \{P'\}\,c_2\,\{Q\}}{\{P\}\,c_1; c_2\,\{Q\}} \qquad \text{assign}\ \frac{}{\{Q[e/x]\}\,x := e\,\{Q\}}$$

$$\text{consequence}\ \frac{P \Rightarrow P' \qquad \{P'\}\,c\,\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\,c\,\{Q\}}$$

The notation $P[e/x]$ means "the logical formula $P$ with every occurrence of variable $x$ replaced by expression $e$." A natural-deduction rule $\frac{A \quad B}{C}$ derives conclusion $C$ from premises $A$ and $B$.

Using these rules, we can derive the validity of the triple $\{a \geq b\}\,(c := a+1;\ b := b-1)\,\{c > b\}$, as follows:

$$\text{seq}\ \frac{\text{con}\ \dfrac{a \geq b \Rightarrow a+1 > b-1 \qquad \text{ass}\ \dfrac{}{\{a+1 > b-1\}\,c := a+1\,\{c > b-1\}}}{\{a \geq b\}\,c := a+1\,\{c > b-1\}} \qquad \text{ass}\ \dfrac{}{\{c > b-1\}\,b := b-1\,\{c > b\}}}{\{a \geq b\}\,(c := a+1;\ b := b-1)\,\{c > b\}}$$

(Here we use a 1-sided version of the rule of consequence, omitting the trivial $c > b-1 \Rightarrow c > b-1$.)

Writing derivation trees in the format above is unwieldy. Hoare-logic proofs can also be presented by interleaving the assertions with the commands; where two assertions appear in a row, the rule of consequence has been used:

assert $\{a \geq b\}$
assert $\{a+1 \geq b-1\}$
c:=a+1;
assert $\{c > b-1\}$
b:=b-1;
assert $\{c > b\}$

MANY OF THE STEPS in deriving a Hoare logic proof can be completely mechanical, with mathematical insight required at only some of the

steps. One useful semiautomatic method is "backward proof", that takes advantage of the way the **assign** rule derives the precondition $Q[e/x]$ from the postcondition $Q$.

Read the following proof from bottom to top:

$\{(a \geq b)\}$ (by mathematics)
$\{(a+1 > b-1)\}$ (by substitution)
$\{(c > b-1)[a+1/c]\}$ (by **assign**)
c:=a+1;
$\{(c > b-1)\}$ (by substitution)
$\{(c > b)[b-1/b]\}$ (by **assign**)
b:=b-1;
$\{c > b\}$ (the given postcondition)

Working backwards, every step labeled "by **assign**" or "by substitution" is completely mechanical; only the step "by mathematics" might require nonmechanical proof—although in this case the proof is easily accomplished by any of several automated semidecision procedures for arithmetic.

SOMETIMES FORWARD PROOF IS NECESSARY. Especially in separation logic (which we will see later), one must establish the a memory-layout precondition before even knowing that a command is safe to execute, so backward proof does not work well. Forward proof can be accomplished with Hoare's assignment rule, but working out the right substition can feel clumsy. Instead we might use Floyd's assignment rule,

$$\text{floyd}\ \frac{}{\{P\}\ x := e\ \{\exists x', x = e[x'/x] \wedge P[x'/x]\}}$$

whose postcondition says, there exists a value $x'$ which is the *old* value of $x$ before the assignment, such that the *new* value of $x$ is the evaluation of expression $e$ but using the old value $x'$ instead of $x$, and the precondition $P$ holds (but again, substituting $x'$ for $x$).

We can try a forward proof of the same program fragment:

$\{(a \geq b)\}$ (the given precondition)
c:=a+1;
$\{\exists c'.\ c = ((a+1)[c'/c]) \ \wedge \ (a \geq b)[c'/c]\}$ (by **floyd**)
$\{c = ((a+1)[c'/c]) \ \wedge \ (a \geq b)[c'/c]\}$ (by $\exists$-elim)
$\{c = a+1 \ \wedge \ (a \geq b)\}$ (by substitution)
b:=b-1;
$\{\exists b'.\ b = ((b-1)[b'/b]) \ \wedge \ (c = a+1 \ \wedge \ a \geq b)[b'/b]\}$ (by **floyd**)
$\{b = b'-1 \ \wedge \ c = a+1 \ \wedge \ a \geq b'\}$ (by $\exists$-elim and substitution)
$\{c > b\}$ (by mathematics)

All the steps except the last are quite mechanical, and the last step is such simple mathematics that many algorithms will also solve it mechanically.

TO REASON ABOUT PROGRAMS WITH CONTROL FLOW, we use the if and while rules.

$$\text{if}\ \frac{\{P \wedge e\}\, c_1\, \{Q\} \qquad \{P \wedge \neg e\}\, c_2\, \{Q\}}{\{P\}\ \text{if}\ e\ \text{then}\ c_1\ \text{else}\ c_2\ \{Q\}} \qquad \text{while}\ \frac{\{I \wedge e\}\, c\, \{I\}}{\{I\}\ \text{while}\ e\ \text{do}\ c\ \{I \wedge \neg e\}}$$

We can use these to prove correctness of an (inefficient) algorithm for division by repeated subtraction. To compute $q = \lfloor a/b \rfloor$, count the number of times $b$ can be subtracted from $a$:

q:=0; **while** (a>b) **do** (a:=a-b; q:=q+1)

To specify this algorithm, we write a precondition and a postcondition; what should they be? We want to say that the quotient $q$ equals $a$ divided by $b$, rounded down. But when the loop is finished, it will *not* be the case that $q = \lfloor a/b \rfloor$, because $a$ has been modified by the loop body. So we make up *auxiliary variables* $a_0, b_0$ to represent the original values of $a$ and $b$. Auxiliary variables are part of the specification or proof but not actually used in the program.

So we might write a precondition $a = a_0 \wedge b = b_0$ and a postcondition $q = \lfloor a_0/b_0 \rfloor$. This looks convincing, but during the proof we will run into trouble if either $a$ or $b$ is negative. This algorithm requires a strengthened precondition, $a = a_0 \wedge b = b_0 \wedge a \geq 0 \wedge b > 0$.

We will use the loop invariant $I = (a_0 = a+bq \wedge b = b_0 \wedge a_0 \geq 0 \wedge b_0 > 0)$. Now the (forward) proof proceeds as follows.

$\{a = a_0 \wedge b = b_0 \wedge a \geq 0 \wedge b > 0\}$
q := 0;
$\{q = 0 \wedge a = a_0 \wedge b = b_0 \wedge a \geq 0 \wedge b > 0\}$
$\{I\}$
**while** (a≥b) **do** (
  $\{a \geq b \wedge I\}$
  $\{a \geq b \wedge a_0 = a + bq \wedge b = b_0 \wedge a_0 \geq 0 \wedge b_0 > 0\}$
   a:=a-b;
  $\{a = a' - b \wedge a' \geq b \wedge a_0 = a' + bq \wedge b = b_0 \wedge a_0 \geq 0 \wedge b_0 > 0\}$
   q:=q+1
  $\{q = q' + 1 \wedge a = a' - b \wedge a' \geq b \wedge a_0 = a' + bq' \wedge b = b_0 \wedge a_0 \geq 0 \wedge b_0 > 0\}$
  $\{a_0 = a + bq \wedge b = b_0 \wedge a_0 \geq 0 \wedge b_0 > 0\}$ (2)
  $\{I\}$
)
$\{I \wedge \neg(a \geq b)\}$
$\{a_0 = a + b_0 q \wedge a_0 \geq 0 \wedge b_0 > 0 \wedge a < b_0\}$
$\{q = \lfloor a_0/b_0 \rfloor\}$ (3)

The only nonmechanical steps in this proof are (1) finding the right loop invariant $I$, and the two rule-of-consequence steps labeled (2) and (3).

It turns out that this algorithm also computes the remainder in variable $a$, so we could have easily proved a stronger postcondition, $a_0 = qb_0 + a \wedge 0 \leq a < b_0$ characterizing the quotient $q$ and remainder $a$.

That algorithm runs in time proportional to $a/b$, which is exponential in the size of the binary representation of $a$.

A more efficient algorithm is *long division*, in which we first shift the divisor $b$ left enough bits until it is greater than $a$, and then repeatedly subtract $z$ from $a$, shifting right after each subtraction. This is a linear time algorithm, assuming that each primitive addition or subtraction takes constant time. It relies on the ability to shift $z$ right by one bit, which we write as $z := z/2$.

$\{a \geq 0 \wedge b > 0\}$
n:=0;
z:=b;
**while** (z≤a)
  **do** (n:=n+1; z:=z+z);
q:=0; r:=a;
**while** (n>0) **do** (
  n:=n-1;
  z:=z/2;
  q:=q+q;
  **if** (z≤r)
  **then** (q:=q+1; r:=r-z)
  **else skip**
)
$\{a = qb + r \wedge 0 \leq r \leq b\}$

This algorithm is complex enough that it really is useful to have a proof of correctness. The precondition and postconditions are shown here. We avoid the need to mention $a_0$ and $b_0$ because the algorithm never assigns to $a$ and $b$, so of course $a = a_0 \wedge b = b_0$. One could prove this formally by adding $a = a_0 \wedge b = b_0$ to both the precondition and the postcondition.

There are two loops here, and their invariants are,

$$I_0 \;=\; z = b2^n \wedge n \geq 0 \wedge a \geq 0 \wedge b > 0$$

$$I_1 \;=\; a = qz + r \wedge 0 \leq r < z \wedge z = b2^n \wedge n \geq 0$$

The reader is invited to work though the steps of the proof, or to consult the detailed proof by Reynolds [81].

# *Chapter 3*

# *Separation logic*

In Hoare logic it is difficult to reason about mutable data structures such as arrays and pointers. One can model the statement $a[i] := v$ as an assignment to $a$ of a new array value, update$(a, i, v)$, such that update$(a, i, v)[i] = v$ and update$(a, i, v)[j] = a[j]$ for $j \neq i$. One cannot simply treat $a[i]$ as a local variable, because assertion $P$ may contain references such as $a[j]$ that may or may not refer to $a[i]$. Instead, one can use a variant of the Hoare assignment rule to model array update:

$$\text{Hoare-array-assign}\ \frac{}{\{P[\text{update}(a, i, v)/a]\}\ a[i] := v\ \{P\}}$$

But this is clumsy: it looks like a global update to all of $a$, instead of a local update to just one slot. For example, consider this judgment:

$$\{a[i] = 5 \wedge a[j] = 7\}\ a[i] := 8\ \{a[i] = 8 \wedge a[j] = 7\}$$

To prove this we "simply" apply the Hoare array-assignment rule and the rule of consequence:

$$\begin{aligned} \text{let } P = &\quad a[i] = 5 \wedge a[j] = 7 \\ Q = &\quad \text{update}(a, i, 8)[i] = 8 \wedge \text{update}(a, i, 8)[j] = 7 \\ R = &\quad a[i] = 8 \wedge a[j] = 7 \end{aligned}$$

$$\text{consequence}\ \frac{P \Rightarrow Q \qquad \text{H-a-a}\ \dfrac{}{\{Q\}\, a[i] := 8\, \{R\}}}{\{P\}\, a[i] := 8\, \{R\}}$$

Proving $P \Rightarrow Q$ requires keeping track of the fact that $i \neq j$ so that we can calculate $(\text{update}(a,i,8))[j] = a[j]$. But wait! We are not told $i \neq j$, so this step is invalid. The correct precondition should have been, $i \neq j \wedge a[i] = 5 \wedge a[j] = 7$.

This illustrates the difficulty: a proliferation of antialiasing facts ($i \neq j$) and tedious rewritings ($i \neq j \Rightarrow \text{update}(a,i,v)[j] = a[j]$). Modeling the pointer update $p.f := v$, on similar principles, is even more clumsy: it looks like a global update to the entire heap.

THE IDEA OF SEPARATION LOGIC is to better support the principle of *local action*. An assertion (precondition or postcondition) holds on a particular *subheap*, or *heaplet*. In Hoare logic we might say $\{P \wedge R\}\, c\, \{Q \wedge R\}$ to mean that $P$ and $R$ both hold on the initial state, $Q$ and $R$ both hold on the final state. In separation logic we say $\{P * R\}\, c\, \{Q * R\}$, meaning that the initial state comprises two disjoint heaplets satisfying $P$ and $R$, and the final state comprises two disjoint heaplets satisfying $Q$ and $R$.

One can think of an assertion $P$ as describing a certain set of addresses, and characterizing the values stored there. The "maps-to" assertion $p \mapsto e$ describes a single-word heaplet whose domain is just address $p$, and says that the value $e$ is stored there. The expression $p$ must be an *l*-value, an expression of the programming language that can appear to the left of an assignment statement. For example, $a[i]$ is an *l*-value in,

$$\{a[i] \mapsto 5 * a[j] \mapsto 7\}\ \ a[i] := 8\ \ \{a[i] \mapsto 8 * a[j] \mapsto 7\}$$

The assertion $a[i] \mapsto 5 * a[j] \mapsto 7$ means that $a[i] \mapsto 5$ and $a[j] \mapsto 7$ hold on two disjoint parts of the heap, and therefore $i \neq j$.

INFERENCE RULES OF SEPARATION LOGIC include the Hoare rules *assignment, sequence, if, consequence* exactly as written on page 11. But we must now understand that each assertion characterizes a particular subheap of the global heap. Furthermore, expressions $e$ can refer only to local variables; they cannot refer to the heap at all. That is, the assignment rule can describe $x := y + z$ but it *does not cover* $x := a[i]$; and assertions can describe $x > y + z$ but *cannot say* $a[i] = v$.

Instead of the assignment rule, we use a *load* rule to fetch $a[i]$, and the *maps-to* assertion $a[i] \mapsto v$. The existential $\exists x'$ in the load rule serves the same purpose as in the Floyd assignment rule (page 12).

$$\text{load-array}\ \frac{}{\{a[e_1] \mapsto e_2\}\, x := a[e_1]\, \{\exists x'.\, x = a[e_1[x'/x]] \wedge (a[e_1] \mapsto e_2)[x'/x]\}}$$

$$\text{store-array}\ \frac{}{\{a[e] \mapsto e_0\}\, a[e] := e_1\, \{a[e] \mapsto e_1\}}$$

$$\text{frame}\ \frac{\{P\}\, c\, \{Q\} \qquad \mathrm{modv}(c) \cap \mathrm{fv}(R) = \emptyset}{\{P * R\}\, c\, \{Q * R\}}$$

THE FRAME RULE IS THE VERY ESSENCE of separation logic. The triple $\{P\}\, c\, \{Q\}$ depends *only* on the part of the heap described by $P$, and modifies *only* that part of the heap (into some state described by $Q$). Any other part of the heap—such as the part described by $R$—is unchanged by the command $c$. In contrast, in an ordinary Hoare logic with ordinary conjuction $\wedge$, the triple $\{P\}\, c\, \{Q\}$ does *not* imply $\{P \wedge R\}\, c\, \{Q \wedge R\}$ (where $P$ and $R$ describe the *same* heap). It is for this reason that separation-logic proofs are more modular than Hoare-logic proofs.

The condition $\mathrm{modv}(c) \cap \mathrm{fv}(R) = \emptyset$ states that the modified variables of the command $c$ must be disjoint from the free variables of the assertion $R$. The command $a[i] := 8$ modifies no variables—storing into one slot of array $a$ does not modify the value of $a$ considered as an address.

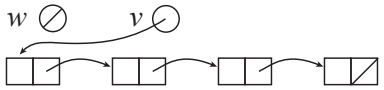
The proof of our array store $a[i] := 8$ is then,

$$\text{frame}\ \frac{\text{store-array}\ \dfrac{}{\{a[i] \mapsto 5\}\, a[i] := 8\, \{a[i] \mapsto 8\}} \qquad \emptyset \cap \{a, i\} = \emptyset}{\{a[i] \mapsto 5 * a[j] \mapsto 7\}\, a[i] := 8\, \{a[i] \mapsto 8 * a[j] \mapsto 7\}}$$

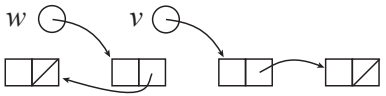
IT IS OBLIGATORY IN AN INTRODUCTION TO SEPARATION LOGIC to present a proof of the in-place list reversal algorithm. Here's some C code that reverses a list (treating 0 as the NULL pointer):

```
/* v points to a linked list */
w=0;
while (v != 0) { t = v.next; v.next = w; w = v; v = t; }
/* w points to the in-place reversal of the list. */
```

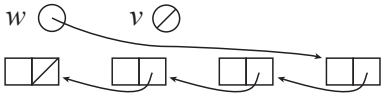
It can be understood using these pictures. At the beginning:




Halfway done:




Done:




Now we prove it in separation logic. The first step is to define what we mean by a list.

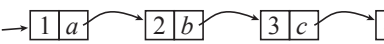
$$\begin{aligned}\mathsf{listshape}(x) = \quad & (x = 0 \wedge \mathsf{emp}) \ \vee \\ & (x \neq 0 \wedge \exists h\, \exists t.\ x.\mathsf{head} \mapsto h * x.\mathsf{next} \mapsto t * \mathsf{listshape}\, t)\end{aligned}$$

That is, $\mathsf{listshape}(x)$ is a recursive predicate, that says either $x$ is nil—the pointer is 0 and the heaplet is empty—or $x$ is the address of a cons cell with head $h$, tail $t$, where $t$ is a list. Furthermore, the head cell is disjoint from all the other list cells. (The predicate emp describes the heap with an empty footprint; it is a unit for the $*$ operator.)

What does that mean, a recursive predicate? There are different choices for the semantics of the recursion operator, as Chapters 10 and 17 will explain. Here we can just use our intuition.

Program analyses in separation logic often want to reason not about the whole list from $x$ to nil, but with *list segments*. The segment from $x$ to $y$ is either empty ($x = y$) or has a first element at address $x$ and has a last element whose tail-pointer is $y$. Written as a recursive predicate, this is,

$$\begin{aligned}\mathsf{listsegshape}(x, y) = \quad & (x = y \wedge \mathsf{emp}) \ \vee \\ & (x \neq y \wedge \exists h\, \exists t.\ x.\mathsf{head} \mapsto h * x.\mathsf{next} \mapsto t \\ & \qquad * \mathsf{listsegshape}(t, y))\end{aligned}$$

For example, the list $p \rightarrow$  contains the segments $p \rightsquigarrow 0$ (the whole list with contents [1,2,3,4]), $p \rightsquigarrow c$ (the segment

with contents $[1,2,3]$), $a \rightsquigarrow c$ (the segment with contents $[2,3]$), $b \rightsquigarrow b$ (an empty segment with contents $[]$), and so on. When we use 0 to represent nil, then $\mathsf{listsegshape}(x,0)$ is the same as $\mathsf{listshape}(x)$.

Some proofs of programs focus on *shape* and *safety*—proving that data structures have the right shape (list? tree? dag? cyclic graph?) and that programs do not dereference nil (or otherwise crash). But sometimes we want proofs of stronger correctness properties. In the case of list-reverse, we may want to prove not only that the result is a list, but that the elements now appear in the reverse order. For such proofs we need to relate the *linked-list* data structure to abstract mathematical *sequences*.

Instead of an operator $\mathsf{listsegshape}(x)$ saying that $x$ points to a list segment, we want to say $\mathsf{listrep}(\sigma)(x,y)$ meaning that $x$ points to a list segment ending at $y$, and the contents (head elements) of that segment are the sequence $\sigma$. That is, the chain of list cells $x \rightsquigarrow y$ is the *representation in memory* of $\sigma$.

$$\begin{aligned}\mathsf{listrep}\,\sigma\,(x,y) = \quad & (x = y \wedge \sigma = \epsilon \wedge \mathsf{emp}) \\ & \vee (x \neq y \wedge \exists \sigma'\, \exists h\, \exists t.\, \sigma = h \cdot \sigma' \\ & \qquad \wedge\ x.\mathsf{head} \mapsto h * x.\mathsf{next} \mapsto t * \mathsf{listrep}\,\sigma'\,(t,y))\end{aligned}$$

We will notate $\mathsf{listrep}\,\sigma\,(x,y)$ as $x \overset{\sigma}{\rightsquigarrow} y$.

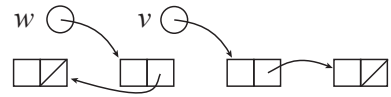
Now we are ready to prove the list-reversal program. The precondition is that $v$ is a linked list representing $\sigma$, and the postcondition is that $w$ represents $\mathrm{rev}(\sigma)$:

$\mathsf{assert}\{v \overset{\sigma}{\rightsquigarrow} 0\}$
$\mathsf{w{=}0;}$ **while** $(\mathsf{v} \mathrel{!}= 0)$ $\{\mathsf{t = v.next;\ v.next = w;\ w = v;\ v = t;}\}$
$\mathsf{assert}\{w \overset{\mathrm{rev}\,\sigma}{\rightsquigarrow} 0\}$

As usual in Hoare logic, we need a loop invariant:

$$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \ \wedge\ v \overset{\sigma_2}{\rightsquigarrow} 0 * w \overset{\sigma_1}{\rightsquigarrow} 0$$

This separation-logic formula describes the picture in which the original sequence $\sigma$ can be viewed as the concatenation of some $\sigma_1$ (reversed) and some $\sigma_2$—we use $\cdot$ to denote se-



quence concatenation—and where the list segment from $v$ to nil represents $\sigma_2$, and the list segment from $w$ to nil represents $\sigma_1$.

To prove this program we need the inference rules of separation logic; the ones shown earlier, plus rules for loading/storing of record fields and for manipulating existential quantifiers. The rule for while is just like the Hoare-logic while rule; as usual, all expressions $e$ (including the while-loop condition) must be *pure*, that is, must not load from the heap directly.

$$\text{load-field}\ \frac{}{\{e_1.\mathsf{fld}\mapsto e_2\}\, x := (e_1.\mathsf{fld})\, \{\exists x'.\ x = e_2[x'/x] \wedge (e_1.\mathsf{fld}\mapsto e_2)[x'/x]\}}$$

$$\text{store-field}\ \frac{}{\{e.\mathsf{fld}\mapsto e_0\}\, (e.\mathsf{fld}) := e_1\, \{e.\mathsf{fld}\mapsto e_1\}}$$

$$\text{while}\ \frac{\{e \wedge P\}\, c\, \{P\}}{\{P\}\, \mathsf{while}\ e\ \mathsf{do}\ c\, \{\neg e \wedge P\}}$$

$$\text{generalize-exists}\ \frac{}{P \vdash \exists x.P} \qquad \text{extract-exists}\ \frac{\{P\}\, c\, \{Q\}}{\{\exists x.P\}\, c\, \{\exists x.Q\}}$$

Our rules for the existential are written in a semiformal ("traditional") mathematical style, assuming that $x$ may be one of the free variables of a formula $P$. In later chapters we will treat this more formally.

Figure 3.1 presents the program annotated with assertions, where each assertion leads to the next. The proof is longer than the program! Checking such a proof by hand might miss some errors. Automating the application of separation logic in a proof assistant ensures that there are no gaps in the proof. Better yet, perhaps parts of the *construction,* not just the *checking,* can be automated.

Let us examine some of the key points in the proof. Just before the while loop (line 3), we have $\{\mathbf{w} = 0 \wedge \mathbf{v} \overset{\sigma}{\rightsquigarrow} 0\}$, that is, the initialization of $w$ and the program precondition that the sequence $\sigma$ is represented by the list starting at pointer v. We must establish the loop invariant (line 5), $\{\exists \sigma_1, \sigma_2.\ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0\}$. To do this we let $\sigma_1$ be the empty sequence and $\sigma_2 = \sigma$.

1 assert{$\mathbf{v} \overset{\sigma}{\rightsquigarrow} 0$}
2 w=0;
3 assert{$\mathbf{w} = 0 \wedge \mathbf{v} \overset{\sigma}{\rightsquigarrow} 0$}
4 assert{let $\sigma_1 = \epsilon,\ \sigma_2 = \sigma$ in $\sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
5 assert{$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
6 **while** (v != 0) with loop invariant{$\exists \sigma_1, \sigma_2.\ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
7 { assert{$\mathbf{v} \neq 0 \wedge \exists \sigma_1, \sigma_2.\ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
8 assert{$\mathbf{v} \neq 0 \wedge \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
9 assert{$\exists \rho, h, p.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) \wedge \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto p * p \overset{\rho}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
10 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) \wedge \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto p * p \overset{\rho}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
11 t = v.next;
12 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) \wedge \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto \mathbf{t} * \mathbf{t} \overset{\rho}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
13 v.next = w;
14 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) \wedge \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto \mathbf{w} * \mathbf{t} \overset{\rho}{\rightsquigarrow} 0 * w \overset{\sigma_1}{\rightsquigarrow} 0$}
15 assert{$\exists q.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) \wedge \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto q * \mathbf{t} \overset{\rho}{\rightsquigarrow} 0 * q \overset{\sigma_1}{\rightsquigarrow} 0$}
16 assert{$\sigma = \mathrm{rev}(h \cdot \sigma_1) \cdot \rho \wedge \mathbf{t} \overset{\rho}{\rightsquigarrow} 0 * \mathbf{v} \overset{h \cdot \sigma_1}{\rightsquigarrow} 0$}
17 assert{$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{t} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{v} \overset{\sigma_1}{\rightsquigarrow} 0$}
18 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{t} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{v} \overset{\sigma_1}{\rightsquigarrow} 0$}
19 w = v;
20 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{t} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
21 v = t;
22 assert{$\sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
23 }
24 assert{$\mathbf{v} = 0 \wedge \exists \sigma_1, \sigma_2. \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0 * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
25 assert{$\exists \sigma_1, \sigma_2.\ \ \sigma = \mathrm{rev}(\sigma_1) \cdot \sigma_2 \wedge \sigma_2 = \epsilon \wedge \mathsf{emp} * \mathbf{w} \overset{\sigma_1}{\rightsquigarrow} 0$}
26 assert{$\mathbf{w} \overset{\mathrm{rev}\,\sigma}{\rightsquigarrow} 0$}

Figure 3.1: List reverse. 4: $\mathrm{rev}(\epsilon) \cdot \sigma = \epsilon \cdot \sigma = \sigma$ 5: by *generalize-exists* 7: by *while* 8: by *extract-exists* 9: by unfolding $\mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0$, then removing the disjunct inconsistent with $v \neq 0$. 10: by *extract-exists* 12: by *load-field*, then eliminating variable $p$ 14: by *store-field* 15: by *generalize-exists* 16: $\mathrm{rev}(\sigma_1) \cdot (h \cdot \rho) = \mathrm{rev}(h \cdot \sigma_1) \cdot \rho$, then fold the definition of $\mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0$ 17: by *generalize-exists* 18: by *extract-exists* 20: by *assign* 22: by *assign* 24: by *while* 25: by folding the definition of $\mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0$, given $\mathbf{v} = 0$ 26: by *extract-exists*, $\mathsf{emp} * P = P$, $\mathrm{rev}(\epsilon) \cdot \sigma_2 = \sigma_2$, then discarding inconsistent conjuncts.

First thing inside the loop body (line 7), we have the loop invariant and the additional fact $\mathsf{v} \neq 0$, and we must *rearrange* the assertion to *isolate* a conjunct of the form $\mathbf{v}.\mathsf{next} \mapsto p$ (at line 10), so that we can load from $\mathsf{v}.\mathsf{next}$. (Both *rearrange* and *isolate* are technical terms in the symbolic execution of programs in separation logic—see Chapter 46.) The loop invariant says that $\sigma_1$ and $\sigma_2$ exist, so we instantiate them using the *extract-exists* rule.

Then (line 9) we can unfold the definition of list-segment $\mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0$; but the nil case is inconsistent with $\mathbf{v} \neq 0$ so we can eliminate it. The non-nil case of $\mathbf{v} \overset{\sigma_2}{\rightsquigarrow} 0$ is

$$(\mathbf{v} \neq 0 \wedge \exists \sigma' \exists h \exists t.\, \sigma = h \cdot \sigma' \wedge\ \mathbf{v}.\mathsf{head} \mapsto h * \mathbf{v}.\mathsf{next} \mapsto t * \mathsf{listrep}\,\sigma\,(t, 0))$$

so we use that, extracting $\sigma', h, t$ by *extract-exists*. Now we can rearrange this assertion into $\mathbf{v}.\mathsf{next} \mapsto p * \textit{other stuff}$, which serves as the precondition for the load rule.

At line 14 after the store command, we have $\mathrm{rev}(\sigma_1) \cdot (h \cdot \rho)$. By the algebra of sequence-reversal and concatenation, this is equivalent to $\mathrm{rev}(h \cdot \sigma_1) \cdot \rho$. We will let the *new* $\sigma_1$ be $h \cdot \sigma_1$, and the *new* $\sigma_2$ be $\rho$. We can fold the definition of $\mathbf{v} \overset{h \cdot \sigma_1}{\rightsquigarrow} 0$ to make (in effect) the same change at the representation level.

The extract-exists rule justifies the transition from line 17 to line 18, that is, from $\mathsf{assert}(\exists \sigma_1.P)$ to $\mathsf{assert}(P)$. It's not that $\exists \sigma_1.P$ entails $P$ in isolation, it's that the Hoare triple $\{\exists \sigma_1.P\}\, c\, \{Q\}$ is provable from $\{P\}\, c\, \{Q\}$. In this case, the Hoare triple $\{\exists \sigma_1.\exists \sigma_2.P\}\, \mathsf{w} = \mathsf{v}; \mathsf{v} = \mathsf{t}\, \{Q\}$ (lines 17–22) is provable from $\{P\}\, \mathsf{w} = \mathsf{v}; \mathsf{v} = \mathsf{t}\, \{Q\}$ (lines 18–22).

At line 24 after the loop, we have the loop invariant *and* the fact that the loop condition is false, therefore $\mathbf{v} = 0$. We can extract the existentially quantified $\sigma_1$ and $\sigma_2$, then notice that $\mathbf{v} = 0$ implies $\sigma_2$ is empty. Thus $\sigma = \mathrm{rev}(\sigma_1)$, and we're done.

For a longer tutorial on separation logic, the reader might try Reynolds [80] or O'Hearn [72].

ARE THE AXIOMS OF SEPARATION LOGIC, as presented in this chapter, really sound—especially when applied to a real programming language, not an idealized one? Building their soundness proof within a proof assistant will

ensure that when we prove properties of programs using separation logic, those properties really hold of their execution. Can this separation logic be used to reason about function-pointers or concurrent threads? All such questions are the subject of the rest of this book.