Chapter 3

Specifications

From assertions, we move on to specifications, which describe the behavior of commands. In this chapter, we will define the syntax and meaning of specifications, give and illustrate inference rules for proving valid specifications, and define a compact form of proof called an "annotated specification".

Since separation logic has been built upon it, we will review the basics of Hoare logic. Further descriptions of this logic, including many examples of proofs, have been given by the author [33, Chapters 1 and 2], [37, Chapters 3 and 4]. A more theoretical view appears in [38, Chapter 8].

The original papers by Hoare [3, 4], as well as earlier work by Naur [39] and Floyd [40], are still well worth reading.

3.1 Hoare Triples

For most of this book, the only kind of specification will be the *Hoare triple*, which consists of two assertions surrounding a command. More precisely, there are two forms of Hoare triple.

A *partial correctness specification*, written

$$\{p\}\ c\ \{q\}$$

is *valid* iff, starting in any state in which the assertion p holds, no execution of the command c aborts and, for any execution of c that terminates in a final state, the assertion q holds in the final state.

A *total correctness specification*, written

$$[\,p\,]\ c\ [\,q\,]$$

is *valid* iff, starting in any state in which p holds, no execution of c aborts, every execution of c terminates, and, for any execution of c that terminates in a final state, q holds in the final state. (In this book, we will consider total specifications infrequently.)

In both forms, p is called the *precondition* (or *precedent*) and q is called the *postcondition* (or *consequent*).

Notice that, in both forms, there is an implicit universal quantification over both initial and final states. Thus the meaning of a specification is simply **true** or **false**, and does not depend upon a state. So to say that a specification is true is the same as saying that it is valid. (This situation will change later when we introduce nonassignable variables denoting procedures and assertions.)

The following are examples of valid partial correctness specifications of simple commands:

$$\{\mathsf{x} - \mathsf{y} > 3\}\ \mathsf{x} := \mathsf{x} - \mathsf{y}\ \{\mathsf{x} > 3\}$$

$$\{\mathsf{x} + \mathsf{y} \geq 17\}\ \mathsf{x} := \mathsf{x} + 10\ \{\mathsf{x} + \mathsf{y} \geq 27\}$$

$$\{\mathbf{emp}\}\ \mathsf{x} := \mathbf{cons}(1, 2)\ \{\mathsf{x} \mapsto 1, 2\}$$

$$\{\mathsf{x} \mapsto 1, 2\}\ \mathsf{y} := [\mathsf{x}]\ \{\mathsf{x} \mapsto 1, 2 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1, 2 \wedge \mathsf{y} = 1\}\ [\mathsf{x} + 1] := 3\ \{\mathsf{x} \mapsto 1, 3 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \mapsto 1, 3 \wedge \mathsf{y} = 1\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x} + 1 \mapsto 3 \wedge \mathsf{y} = 1\}$$

$$\{\mathsf{x} \leq 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\}$$

$$\{\mathbf{true}\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathsf{x} = 10\} \quad (*)$$

$$\{\mathsf{x} > 10\}\ \mathbf{while}\ \mathsf{x} \neq 10\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1\ \{\mathbf{false}\} \quad (*)$$

A more elaborate example is a specification of the "record-gluing" program (1.5):

$$\begin{array}{l}
\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\mathbf{if}\ \mathsf{y} = \mathsf{x} + 1\ \mathbf{then\ skip} \\
\quad \mathbf{else\ if}\ \mathsf{x} = \mathsf{y} + 1\ \mathbf{then}\ \mathsf{x} := \mathsf{y} \\
\quad \mathbf{else}\ (\mathbf{dispose}\ \mathsf{x}\ ; \mathbf{dispose}\ \mathsf{y}\ ; \mathsf{x} := \mathbf{cons}(1, 2)) \\
\{\mathsf{x} \mapsto -, -\}
\end{array}$$

(All of the above examples, except those marked $(*)$, would also be valid as total specifications.)

3.2 Hoare's Inference Rules for Specifications

As we did with assertions in Section 2.2, we will reason about specifications using inference rules of the form

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\mathcal{C},}$$

where the premisses and conclusion may contain metavariables, each of which ranges over some set of phrases, such as expressions, variables, or assertions. Again, a rule is said to be *sound* iff, for every instance of the rule, the conclusion of the instance is valid whenever all of the premisses of the instance are valid. But now the premisses may be either specifications or assertions, and the conclusion must be a specification.

Here we give the original rules of Hoare logic [3] (along with a more recent rule for variable declarations), all of which remain valid for separation logic. In each case, we give the rule and one or more of its instances:

- Assignment (AS)

$$\frac{}{\{p/v \to e\}\ v := e\ \{p\}}$$

Instances:

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

- Sequential Composition (SQ)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;c_2\ \{r\}}$$

An instance:

$$\frac{\begin{array}{r}\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \qquad\qquad \\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\ ;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

- Strengthening Precedent (SP)

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\ c\ \{r\}}$$

An instance:

$$\frac{\begin{array}{l}\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n} \\ \qquad \{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

In contrast to rules such as Assignment and Sequential Composition, which are called *command-specific rules*, the rules for Strengthening Precedents and Weakening Consequents (to be introduced later) are applicable to arbitrary commands, and are therefore called *structural rules*.

These two rules are exceptional in having premisses, called *verification conditions*, that are assertions rather than specifications. The verification conditions are the mechanism used to introduce mathematical facts about kinds of data, such as $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}$, into proofs of specifications.

To be completely formal, our notion of proof should include formal subproofs of verification conditions, using the rules of predicate calculus as well as rules about integers and other kinds of data. In this book. however, to avoid becoming mired in proofs of simple arithmetic facts, we will often omit the proofs of verification conditions. It must be emphasized, however, that the soundness of a formal proof can be destroyed by an invalid verification condition.

- Partial Correctness of **while** (WH)

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i \wedge \neg\, b\}}$$

An instance:

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k}+1\,;\mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\}\end{array}}$$

Here the assertion i is the *invariant* of the **while** command. This is the one inference rule in this chapter that does not extend to total correctness — reflecting that the **while** command is the one construct in our programming language that can cause nontermination. Later, however, we will find a similar situation with the rule for recursive procedures.

- Weakening Consequent (WC)

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\ c\ \{r\}}$$

An instance:

$$\frac{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \\ \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}\end{array}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Notice that $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}$ is another verification condition.

At this stage, we can give a slightly nontrivial example of a formal proof (of the heart of a simple program for computing powers of two):

1. $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}$ (VC)
2. $\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (AS)
3. $\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (AS)
4. $\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
   (SQ 2,3)
5. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
   (SP 1,4)
6. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WH 5)
   while $\mathsf{k} \neq \mathsf{n}$ **do** $(\mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y})$
   $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n}\}$
7. $\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}$ (VC)
8. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WC 6,7)
   while $\mathsf{k} \neq \mathsf{n}$ **do** $(\mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y})$
   $\{\mathsf{y} = 2^{\mathsf{n}}\}$

Additional rules describe additional forms of commands:

- **skip** (SK)

$$\frac{}{\{p\}\ \textbf{skip}\ \{p\}}$$

An instance:

$$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\ \textbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}$$

- Conditional (CD)

$$\frac{\{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \neg\ b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

An instance:

$$\frac{\begin{array}{c}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})\}\ \mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\ \mathbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\end{array}}{\{\mathsf{y} = 2^{\mathsf{k}}\}\ \mathbf{if}\ \mathrm{odd}(\mathsf{k})\ \mathbf{then}\ \mathsf{k} := \mathsf{k} + 1; \mathsf{y} := 2 \times \mathsf{y}\ \mathbf{else\ skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}}$$

- Variable Declaration (DC)

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ \mathbf{newvar}\ v\ \mathbf{in}\ c\ \{q\}}$$

when v does not occur free in p or q.

An instance:

$$\frac{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\}\\ \mathsf{k} := 0\,; \mathsf{y} := 1\,;\\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,; \mathsf{y} := 2 \times \mathsf{y})\\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\}\\ \mathbf{newvar}\ \mathsf{k}\ \mathbf{in}\\ \quad\big(\mathsf{k} := 0\,; \mathsf{y} := 1\,;\\ \quad\mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,; \mathsf{y} := 2 \times \mathsf{y})\big)\\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Here the requirement on the declared variable v formalizes the concept of *locality*, i.e., that the value of v when c begins execution has no effect on this execution, and that the value of v when c finishes execution has no effect on the rest of the program.

Notice that the concept of locality is context-dependent: Whether a variable is local or not depends upon the requirements imposed by the surrounding program, which are described by the specification that reflects these requirements. For example, in the specification

$$\{\mathbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\,; \mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2\},$$

t is local, and can be declared at the beginning of the command being specified, but in

$$\{\mathbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\,;\mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2 \wedge \mathsf{t} = (\mathsf{x} + \mathsf{y})\},$$

t is not local, and cannot be declared.

For several of the rules we have given, there are alternative versions. For instance:

- Alternative Rule for Assignment (ASalt)

$$\frac{}{\{p\}\ v := e\ \{\exists v'.\ v = e' \wedge p'\}}$$

where $v' \notin \{v\} \cup \mathrm{FV}(e) \cup \mathrm{FV}(p)$, e' is $e/v \to v'$, and p' is $p/v \to v'$. The quantifier can be omitted when v does not occur in e or p.

The difficulty with this "forward" rule (due to Floyd [40]) is the accumulation of quantifiers, as in the verification condition in the following proof:

1. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n}\}$ (ASalt)
2. $\{\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}$ (ASalt)
   $\{\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})\}$
3. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}$ (SQ 1)
   $\{\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})\}$
4. $(\exists \mathsf{y}'.\ \mathsf{y} = 2 \times \mathsf{y}' \wedge (\exists \mathsf{k}'.\ \mathsf{k} = \mathsf{k}' + 1 \wedge \mathsf{y}' = 2^{\mathsf{k}'} \wedge \mathsf{k}' < \mathsf{n})) \Rightarrow (\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n})$
   (VC)
5. $\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} < \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\,;\mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ (WC 3,4)

(Compare Step 4 with the verification condition in the previous formal proof.)

- Alternative Rule for Conditionals (CDalt)

$$\frac{\{p_1\}\ c_1\ \{q\} \qquad \{p_2\}\ c_2\ \{q\}}{\{(b \Rightarrow p_1) \wedge (\neg\ b \Rightarrow p_2)\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \{q\}}$$

An instance:

$$\frac{\begin{array}{c}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\ \mathbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\end{array}}{\begin{array}{l}\{(\mathrm{odd}(\mathsf{k}) \Rightarrow \mathsf{y} = 2^{\mathsf{k}} \wedge \mathrm{odd}(\mathsf{k})) \wedge (\neg\,\mathrm{odd}(\mathsf{k}) \Rightarrow \mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k}))\}\\ \mathbf{if}\ \mathrm{odd}(\mathsf{k})\ \mathbf{then}\ \mathsf{k} := \mathsf{k} + 1; \mathsf{y} := 2 \times \mathsf{y}\ \mathbf{else\ skip}\\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathrm{odd}(\mathsf{k})\}\end{array}}$$

(A comparison with the instance of (CD) given earlier indicates why (CD) is usually preferable to (CDalt).)

There is also a rule that combines the rules for strengthening precedents and weakening consequences:

- Consequence (CONSEQ)

$$\frac{p \Rightarrow p' \qquad \{p'\}\ c\ \{q'\} \qquad q' \Rightarrow q}{\{p\}\ c\ \{q\}}$$

An instance:

$$\frac{\begin{array}{l}(\mathsf{n} \geq 0) \Rightarrow (1 = 2^0 \wedge 0 \leq \mathsf{n})\\[4pt] \{1 = 2^0 \wedge 0 \leq \mathsf{n}\}\\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ;\\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n}\}\\[4pt] (\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\,\mathsf{k} \neq \mathsf{n}) \Rightarrow (\mathsf{y} = 2^{\mathsf{n}})\end{array}}{\begin{array}{l}\{\mathsf{n} \geq 0\}\\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ;\\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

(One can use (CONSEQ) in place of (SP) and (WC), but the price is an abundance of vacuous verification conditions of the form $p \Rightarrow p$.)

3.3 Annotated Specifications

As was made evident in the previous section, full-blown formal proofs in Hoare logic, even of tiny programs, are long and tedious. Because of this, the usual way of presenting such proofs, at least to human readers, is by means of a specification that has been annotated with intermediate assertions in addition to its pre- and postconditions. Several examples of such *annotated specifications* (also often called "proof outlines") have already appeared in Sections 1.5 and 1.10.

The purpose of an annotated specification is to provide enough information so that it would be straightforward for the reader to construct a full formal proof, or at least to determine the verification conditions occuring in the proof. In this section, we will formulate rigorous criteria for achieving this purpose.

In the first place, we note that without annotations, it is not straightforward to construct a proof of a specification from the specification itself. For example, if we try to use the rule for sequential composition,

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ c_2\ \{r\},}$$

to obtain the main step of a proof of the specification

$$\begin{array}{l}\{\mathsf{n} \geq 0\}\\ (\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ;\\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\\ \{\mathsf{y} = 2^{\mathsf{n}}\},\end{array}$$

there is no indication of what assertion should replace the metavariable q.

But if we change the rule to

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ \underline{\{q\}}\ c_2\ \{r\},}$$

then the new rule requires the annotation q (underlined here for emphasis) to occur in the conclusion:

$\{\mathsf{n} \geq 0\}$
$(\mathsf{k} := 0 \,; \mathsf{y} := 1) \,;$
$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
$\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1 \,; \mathsf{y} := 2 \times \mathsf{y})$
$\{\mathsf{y} = 2^{\mathsf{n}}\}.$

Then, once q is determined, the premisses must be

$\{\mathsf{n} \geq 0\}$
$(\mathsf{k} := 0 \,; \mathsf{y} := 1);$
$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$

and

$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
$\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1 \,; \mathsf{y} := 2 \times \mathsf{y})$
$\{\mathsf{y} = 2^{\mathsf{n}}\}.$

The basic trick is to add annotations to the conclusions of the inference rules so that the conclusion of each rule completely determines its premisses. However, it is necessary to distinguish between an annotated specification and the unannotated specification that it proves (since, in a few cases, more complex manipulations than the mere erasure of annotations will be needed to reveal the command being specified). For this purpose, in the inference rules for annotated specifications, we will use premisses (except for verification conditions) and conclusions of the form

$$\{p\}\ \hat{c}\ \{q\} \gg c,$$

which can be read "$\{p\}\ \hat{c}\ \{q\}$ is an annotated specification proving the unannotated specification $\{p\}\ c\ \{q\}$".

In fact, among the rules in the previous section (excluding the alternative rules (ASalt), (CDalt), and (CONSEQ)), only three rules require the insertion of annotations:

- Sequential Composition (SQan)

$$\frac{\{p\}\ \hat{c}_1\ \{q\} \gg c_1 \qquad \{q\}\ \hat{c}_2\ \{r\} \gg c_2}{\{p\}\ \hat{c}_1\ ;\ \{q\}\ \hat{c}_2\ \{r\} \gg c_1\ ;\ c_2}$$

- Strengthening Precedent (SPan)

$$\frac{p \Rightarrow q \qquad \{q\}\ \hat{c}\ \{r\} \gg c}{\{p\}\ \{q\}\ \hat{c}\ \{r\} \gg c}$$

- Weakening Consequent (WCan)

$$\frac{\{p\}\ \hat{c}\ \{q\} \gg c \qquad q \Rightarrow r}{\{p\}\ \hat{c}\ \{q\}\ \{r\} \gg c.}$$

Strictly speaking, all the remaining rules should be rewritten in a uniform way, by replacing $\{p\}\ \cdots c_1 \cdots c_n \cdots\ \{q\}$ by

$$\{p\}\ \cdots \hat{c}_1 \cdots \hat{c}_n \cdots\ \{q\} \gg \cdots c_1 \cdots c_n \cdots.$$

For example, the conditional rule (CD) would become

$$\frac{\{p \wedge b\}\ \hat{c}_1\ \{q\} \gg c_1 \qquad \{p \wedge \neg\, b\}\ \hat{c}_2\ \{q\} \gg c_2}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ \hat{c}_1\ \mathbf{else}\ \hat{c}_2\ \{q\} \gg \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2.}$$

For readability, however, we will not make this change to rules that do not introduce new annotations.

The following is an annotated specification of the program for computing powers of two, which can be derived using the inference rules we have given, including the modified rules above:

$\{\mathsf{n} \geq 0\}$
newvar k **in**
  $\big(\{1 = 2^0 \wedge 0 \leq \mathsf{n}\}$ $\qquad (*)$
  $\mathsf{k} := 0\ ;$
  $\{1 = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$ $\qquad (*)$
  $\mathsf{y} := 1\ ;$

$$
\begin{array}{ll}
\quad\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} & \\
\quad\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do} & \\
\qquad\big(\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\} & (*) \\
\qquad\mathsf{k} := \mathsf{k} + 1\,; & \\
\qquad\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} & (*) \\
\qquad\mathsf{y} := 2 \times \mathsf{y}\big) & \\
\quad\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\}\big) & (**) \\
\{\mathsf{y} = 2^{\mathsf{n}}\} &
\end{array}
$$

(The reader will find it instructive to construct the formal proof of this specification from its annotations — and to determine thereby the verification conditions.)

In fact, the above specification contains more annotations then necessary. For instance, because of the form of the assignment rule,

$$\frac{}{\{p/v \to e\}\ v := e\ \{p\}},$$

the precondition of an assignment $v{:=}e$ is determined by its postcondition, so that the annotations marked $(*)$ in the above specification are unnecessary.

Also, because of the form of the **while** rule,

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{i \wedge \neg\, b\}},$$

the postcondition of a **while** command is determined by its precondition, so that the annotation marked $(**)$ is unnecessary.

To formalize such reductions in annotations, we change three further inference rules. (Each of the new versions can be derived from earlier rules.)

- Assignment (ASan)

$$\frac{p_0 \Rightarrow (p/v \to e)}{\{p_0\}\ v := e\ \{p\}} \qquad\qquad \frac{\{p_0\}\ c\ \{p/v \to e\}}{\{p_0\}\ c\,;v := e\ \{p\}}$$

- **skip** (SKan)

$$\frac{p_0 \Rightarrow p}{\{p_0\}\ \textbf{skip}\ \{p\}}$$

- Partial Correctness of **while** (WHan)

$$\frac{\{i \wedge b\}\ c\ \{i\} \qquad (i \wedge \neg\, b) \Rightarrow p}{\{i\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{p\}.}$$

Using these rules, we obtain the following minimally annotated specification:

$$\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
\textbf{newvar}\ \mathsf{k}\ \textbf{in} \\
\quad \big(\mathsf{k} := 0\,;\, \mathsf{y} := 1\,; \\
\quad \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\quad \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\,;\, \mathsf{y} := 2 \times \mathsf{y})\big) \\
\{\mathsf{y} = 2^{\mathsf{n}}\}
\end{array}$$

(The reader will also find it instructive to construct the formal proof of this specification.)

It should be emphasized that the use of the rules (SQan), (SPan), and (WCan) is necessary to obtain annotated specifications that determine their own formal proofs, but the use of the rules (ASan), (SKan), and (WHan) to minimize annotations is optional. We will often provide more annotations than necessary, in order to make a proof clearer.

3.4 More Structural Inference Rules

In addition to Strengthening Precedent and Weakening Consequent, there are a number of other structural inference rules of Hoare logic that remain sound for separation logic. In this section and the next, we will ignore annotated specifications, and give the simpler rules that suffice for formal proofs.

- Vacuity (VAC)

$$\frac{}{\{\textbf{false}\}\ c\ \{q\}}$$

This rule is sound because the definition of both partial and total correctness specifications begin with a universal quantification over states satisfying the precondition — and there are no states satisfying **false**. It is useful for characterizing commands that are not executed (often the bodies of **while** commands). For example, one can prove

1. $(\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \mathsf{k} < \mathsf{b}) \Rightarrow \mathbf{false}$ (VC)
2. $\{\mathbf{false}\}$ (VAC)
   $\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$
3. $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \mathsf{k} < \mathsf{b}\}$ (VAC)
   $\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$
4. $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\}$ (WH)
   $\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k})$
   $\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b} \wedge \neg\, \mathsf{k} < \mathsf{b}\}.$

- Disjunction (DISJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \vee p_2\}\ c\ \{q_1 \vee q_2\}}$$

For example, consider two (annotated) specifications of the same command:

$$\begin{array}{ll}
\{\mathsf{a} - 1 \leq \mathsf{b}\} & \{\mathsf{a} - 1 \geq \mathsf{b}\} \\
\mathsf{s} := 0 \,;\, \mathsf{k} := \mathsf{a} - 1 \,; & \mathsf{s} := 0 \,;\, \mathsf{k} := \mathsf{a} - 1 \,; \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{k}} \mathsf{i} \wedge \mathsf{k} \leq \mathsf{b}\} & \{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\} \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} & \mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}) & \quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\} & \{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b}\}.
\end{array}$$

(The second specification describes the situation where the body of the **while** command is never executed.) Using these specifications as premisses to

(DISJ), we can obtain the main step in

$$\begin{array}{l}
\{\mathbf{true}\} \\
\{\mathsf{a} - 1 \leq \mathsf{b} \vee \mathsf{a} - 1 \geq \mathsf{b}\} \\
\mathsf{s} := 0 \,;\, \mathsf{k} := \mathsf{a} - 1 \,; \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1 \,;\, \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{(\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}) \vee (\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b})\} \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}.
\end{array}$$

- Conjunction (CONJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}}$$

(It should be noted that, in some extensions of separation logic, this rule becomes unsound.)

- Existential Quantification (EQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

where v is not free in c.

- Universal Quantification (UQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\forall v.\ p\}\ c\ \{\forall v.\ q\},}$$

where v is not free in c.

- Substitution (SUB)

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots\ ,\ v_n \to e_n,}$$

where $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

The restrictions on this rule are needed to avoid aliasing. For example, in

$$\{\mathsf{x} = \mathsf{y}\}\ \mathsf{x} := \mathsf{x} + \mathsf{y}\ \{\mathsf{x} = 2 \times \mathsf{y}\},$$

one can substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{w} - 1$ to infer

$$\{\mathsf{z} = 2 \times \mathsf{w} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{w} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{w} - 1)\}.$$

But one cannot substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{z} - 1$ to infer the invalid

$$\{\mathsf{z} = 2 \times \mathsf{z} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{z} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{z} - 1)\}.$$

This rule for substitution will become important when we consider procedures.

- Renaming (RN)

$$\frac{\{p\}\ \textbf{newvar}\ v\ \textbf{in}\ c\ \{q\}}{\{p\}\ \textbf{newvar}\ v'\ \textbf{in}\ (c/v \to v')\ \{q\},}$$

  where v' does not occur free in c.

Actually, this is not a structural rule, since it is only applicable to variable declarations. On the other hand, unlike the other nonstructural rules, it is not syntax-directed.

The only time it is necessary to use this rule is when one must prove a specification of a variable declaration that violates the proviso that the variable being declared must not occur free in the pre- or postcondition. For example,

1. $\{\mathsf{x} = 0\}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}$ (AS)
2. $\{\mathsf{x} = 0\}\ \textbf{newvar}\ \mathsf{y}\ \textbf{in}\ \mathsf{y} := 1\ \{\mathsf{x} = 0\}$ (DC 1)
3. $\{\mathsf{x} = 0\}\ \textbf{newvar}\ \mathsf{x}\ \textbf{in}\ \mathsf{x} := 1\ \{\mathsf{x} = 0\}$. (RN 2)

In practice, the rule is rarely used. It can usually be avoided by renaming local variables in the program before proving it.

Renaming of bound variables in pre- and postconditions can be treated by using verification conditions such as $(\forall \mathsf{x}.\ \mathsf{z} \neq 2 \times \mathsf{x}) \Rightarrow (\forall \mathsf{y}.\ \mathsf{z} \neq 2 \times \mathsf{y})$ as premisses in the rules (SP) and (WC).

3.5 The Frame Rule

In Section 1.5, we saw that the Hoare-logic Rule of Constancy fails for separation logic, but is replaced by the more general

- Frame Rule (FR)

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\},}$$

  where no variable occurring free in r is modified by c.

In fact, the frame rule lies at the heart of separation logic, and provides the key to local reasoning. An instance is

$$\frac{\{\mathbf{list}\ \alpha\ \mathsf{i}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j}\}}{\{\mathbf{list}\ \alpha\ \mathsf{i} * \mathbf{list}\ \gamma\ \mathsf{k}\}\ \text{“Reverse List”}\ \{\mathbf{list}\ \alpha^{\dagger}\ \mathsf{j} * \mathbf{list}\ \gamma\ \mathsf{k}\},}$$

(assuming “Reverse List” does not modify k or γ).

The soundness of the frame rule is surprisingly sensitive to the semantics of our programming language. Suppose, for example, we changed the behavior of deallocation, so that, instead of causing a memory fault, **dispose** x behaved like **skip** when the value of x was not in the domain of the heap. Then $\{\mathbf{emp}\}$ **dispose** x $\{\mathbf{emp}\}$ would be valid, and the frame rule could be used to infer $\{\mathbf{emp} * \mathsf{x} \mapsto 10\}$ **dispose** x $\{\mathbf{emp} * \mathsf{x} \mapsto 10\}$. Then, since **emp** is a neutral element for $*$, we would have $\{\mathsf{x} \mapsto 10\}$ **dispose** x $\{\mathsf{x} \mapsto 10\}$, which is patently false.

To reveal the programming-language properties that the frame rule depends upon, we begin with some definitions:

> If, starting in the state s, h, no execution of a command c aborts, then c is *safe at* s, h. If, starting in the state s, h, every execution of c terminates without aborting, then c *must terminate normally* at s, h.

Then the frame rule depends upon two properties of the programming language, which capture the idea that a program will only depend upon or change the part of the initial heap within its footprint, and will abort if any of that part of the heap is missing:

> *Safety Monotonicity* If $\hat{h} \subseteq h$ and c is safe at $s, h - \hat{h}$, then c is safe at s, h. If $\hat{h} \subseteq h$ and c must terminate normally at $s, h - \hat{h}$, then c must terminate normally at s, h.

The Frame Property If $\hat{h} \subseteq h$, c is safe at $s, h - \hat{h}$, and some execution of c starting at s, h terminates normally in the state s', h', then $\hat{h} \subseteq h'$ and some execution of c starting at $s, h - \hat{h}$, terminates normally in the state $s', h' - \hat{h}$.

Then:

Proposition 11 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

PROOF Assume $\{p\}\ c\ \{q\}$ is valid, and $s, h \models p * r$. Then there is an $\hat{h} \subseteq h$ such that $s, h - \hat{h} \models p$ and $s, \hat{h} \models r$.

From $\{p\}\ c\ \{q\}$, we know that c is safe at $s, h - \hat{h}$ (and in the total-correctness case it must terminate normally). Then, by safety monotonicity, we know that c is safe at s, h (and in the total-correctness case it must terminate normally).

Suppose that, starting in the state s, h, there is an execution of c that terminates in the state s', h'. Since c is safe at $s, h - \hat{h}$, we know by the frame property that $\hat{h} \subseteq h'$ and that, starting in the state $s, h - \hat{h}$, there is some execution of c that terminates in the state $s', h' - \hat{h}$. Then $\{p\}\ c\ \{q\}$ and $s, h - \hat{h} \models p$ imply that $s', h' - \hat{h} \models q$.

Since an execution of c carries s, h into s', h', we know that $s\,v = s'v$ for all v that are not modified by c. Then, since these include the free variables of r, $s, \hat{h} \models r$ implies that $s', \hat{h} \models r$. Thus $s', h' \models q * r$. END OF PROOF

3.6 More Rules for Annotated Specifications

We now consider how to enlarge the concept of annotated specifications to encompass the structural rules given in the two preceding sections. Several general properties hold for all of these rules:

- Since the rules are applicable to arbitrary commands, their annotated versions must indicate which rule is being applied.

- Since the rules are intended for multi-line specifications, they will be given in a vertical or two-dimensional format. In particular, instead of

$\{p\}\ \hat{c}\ \{q\} \gg c$, we will write

$$\begin{array}{l} \{p\} \\ \ \hat{c} \quad \gg c. \\ \{q\} \end{array}$$

- Although the rules (like (SQan), (SPan), and (WCan)) are necessary to obtain annotated specifications that determine their own formal proofs, they also permit the minimization of annotations in much the same way as (ASan), (SKan), and (WHan).

For the Vacuity Rule, it is enough to indicate the rule being applied, as in

$$\begin{array}{l} \{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\} \\ \textbf{while } \mathsf{k} < \mathsf{b} \textbf{ do} \\ \quad (\mathsf{k} := \mathsf{k} + 1\, ;\, \mathsf{s} := \mathsf{s} + \mathsf{k})\Big\}\mathrm{VAC} \\ \{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b}\}. \end{array}$$

This annotation is described by the inference rule:

- Vacuity (VACan)

$$\frac{p \Rightarrow \textbf{false}}{\begin{array}{l} \{p\} \\ \ c \Big\}\mathrm{VAC} \ \gg c \\ \{q\} \end{array}}$$

Here c contains no annotations, since no reasoning about its subcommands is used.

For the two-premiss rules (DISJ) and (CONJ), a perspicuous approach is to display the proofs of the premisses side-by-side. Thus the example of the use of (DISJ) would be conveyed by the annotated specification:

$$
\begin{array}{ccc}
& \{\textbf{true}\} & \\
\begin{array}{l}
\{\mathsf{a}-1 \le \mathsf{b}\} \\
\mathsf{s} := 0\,;\, \mathsf{k} := \mathsf{a}-1\,; \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{k}} \mathsf{i} \wedge \mathsf{k} \le \mathsf{b}\} \\
\textbf{while } \mathsf{k} < \mathsf{b} \textbf{ do} \\
\quad (\mathsf{k} := \mathsf{k}+1\,;\, \mathsf{s} := \mathsf{s}+\mathsf{k}) \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}
\end{array}
& \text{DISJ} &
\begin{array}{l}
\{\mathsf{a}-1 \ge \mathsf{b}\} \\
\mathsf{s} := 0\,;\, \mathsf{k} := \mathsf{a}-1\,; \\
\{\mathsf{s} = 0 \wedge \mathsf{a}-1 \ge \mathsf{b} \wedge \mathsf{k} \ge \mathsf{b}\} \\
\textbf{while } \mathsf{k} < \mathsf{b} \textbf{ do} \\
\quad (\mathsf{k} := \mathsf{k}+1\,;\, \mathsf{s} := \mathsf{s}+\mathsf{k})\}\text{VAC} \\
\{\mathsf{s} = 0 \wedge \mathsf{a}-1 \ge \mathsf{b}\}.
\end{array} \\
& \{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\} \qquad . &
\end{array}
$$

Here it is not necessary to give the precondition $\mathsf{a}-1 \le \mathsf{b} \vee \mathsf{a}-1 \ge \mathsf{b}$ or the postcondition $(\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}) \vee (\mathsf{s} = 0 \wedge \mathsf{a}-1 \ge \mathsf{b})$, since both are determined by the pre- and postconditions of the premisses, along with the identity of the rule (DISJ).

This construction is described by the following inference rule:

- Disjunction (DISJan)

$$
\frac{p \Rightarrow (p_1 \vee p_2) \qquad \begin{array}{c}\{p_1\}\\ \hat{c}_1\\ \{q_1\}\end{array} \gg c \qquad \begin{array}{c}\{p_2\}\\ \hat{c}_2\\ \{q_2\}\end{array} \gg c \qquad (q_1 \vee q_2) \Rightarrow q}{\begin{array}{ccc} & \{p\} & \\ \begin{array}{c}\{p_1\}\\ \hat{c}_1\\ \{q_1\}\end{array} & \text{DISJ} & \begin{array}{c}\{p_2\}\\ \hat{c}_2\\ \{q_2\}\end{array} \\ & \{q\} & \end{array} \gg c}
$$

Similarly, we have:

- Conjunction (CONJan)

$$\frac{p \Rightarrow (p_1 \wedge p_2) \qquad \begin{array}{c}\{p_1\}\\ \hat{c}_1\\ \{q_1\}\end{array} \gg c \qquad \begin{array}{c}\{p_2\}\\ \hat{c}_2\\ \{q_2\}\end{array} \gg c \qquad (q_1 \wedge q_2) \Rightarrow q}{\begin{array}{ccc} & \{p\} & \\ \{p_1\} & & \{p_2\}\\ \hat{c}_1 & \mathrm{CONJ} & \hat{c}_2\\ \{q_1\} & & \{q_2\}\\ & \{q\} & \end{array} \gg c}$$

A similar approach, with indentation instead of a two-column display, is used for the rules for quantification:

- Existential Quantification (EQan)

$$\frac{p' \Rightarrow (\exists v.\, p) \qquad \begin{array}{c}\{p\}\\ \hat{c}\\ \{q\}\end{array} \gg c \qquad (\exists v.\, q) \Rightarrow q'}{\begin{array}{l}\{p'\}\\ \quad \left.\begin{array}{c}\{p\}\\ \hat{c}\\ \{q\}\end{array}\right\} \exists v\\ \{q'\}\end{array} \gg c,}$$

  where v is not free in c.

- Universal Quantification (UQan)

$$\frac{p' \Rightarrow (\forall v.\, p) \qquad \begin{array}{c}\{p\}\\ \hat{c}\\ \{q\}\end{array} \gg c \qquad (\forall v.\, q) \Rightarrow q'}{\begin{array}{l}\{p'\}\\ \quad \left.\begin{array}{c}\{p\}\\ \hat{c}\\ \{q\}\end{array}\right\} \forall v\\ \{q'\}\end{array} \gg c,}$$

  where v is not free in c.

In these rules, the pre-and postconditions $\exists v.\ p$ and $\exists v.\ q$ (or $\forall v.\ p$ and $\forall v.\ q$) need not appear since they are determined by the pre- and postconditions of the premiss, the identity of the quantifier, and the name of the bound variable v.

The same approach works for the frame rule:

- Frame Rule (FRan)

$$\frac{p' \Rightarrow (p \ast r) \qquad \begin{matrix}\{p\}\\ \hat{c}\\ \{q\}\end{matrix} \gg c \qquad (q \ast r) \Rightarrow q'}{\begin{matrix}\{p'\}\\ \left.\begin{matrix}\{p\}\\ \hat{c}\\ \{q\}\end{matrix}\right\} \ast\ r \gg c,\\ \{q'\}\end{matrix}}$$

  where no variable occurring free in r is modified by c.

Here the pre-and postconditions $p \ast r$ and $q \ast r$ are determined by p, q, and r.

Annotating the use of the substitution rule is more difficult. The sensible approach seems to be to indicate the substitution explicitly as an annotation:

- Substitution (SUBan)

$$\frac{p' \Rightarrow (p/\delta) \qquad \begin{matrix}\{p\}\\ \hat{c}\\ \{q\}\end{matrix} \gg c \qquad (q/\delta) \Rightarrow q'}{\begin{matrix}\{p'\}\\ \left.\begin{matrix}\{p\}\\ \hat{c}\\ \{q\}\end{matrix}\right\} /\delta \gg c/\delta,\\ \{q'\}\end{matrix}}$$

  where δ is the substitution $v_1 \to e_1, \ldots, v_n \to e_n$, $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j. The total substitution δ may be abbreviated by a partial substitution, but the conditions on what is substituted for the v_i must hold for all $v_1, \ldots, v_n$ occurring free in p, c, or q.

Notice here that recovering the command being specified by the conclusion can no longer be done by erasing annotations; one must carry out the indicated substitution. A similar situation holds for the Renaming rule:

- Renaming (RNan)

$$\frac{\begin{array}{l}\{p\}\\ \textbf{newvar}\ v\ \textbf{in}\ \hat{c}\ \gg \textbf{newvar}\ v\ \textbf{in}\ c\\ \{q\}\end{array}}{\begin{array}{l}\{p\}\\ \textbf{newvar}\ v\ \textbf{renamed}\ v'\ \textbf{in}\ \hat{c}\ \gg \textbf{newvar}\ v'\ \textbf{in}\ (c/v \to v'),\\ \{q\}\end{array}}$$

  where v' does not occur free in c.

3.7 Inference Rules for Mutation and Disposal

Finally, we come to the new commands for manipulating the heap, which give rise to a surprising variety of inference rules. For each of these four commands, we can give three kinds of inference rule: local, global, and backward-reasoning.

In this and the next two section, we will present these rules and show that, for each type of command, the different rules are all derivable from one another (except for specialized rules that are only applicable in the "nonoverwriting" case). (Some of the deriviations will make use of the inference rules for assertions derived in Section 2.4.)

For mutation commands, we have

- The local form (MUL)

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

- The global form (MUG)

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

- The backward-reasoning form (MUBR)

$$\overline{\{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!\ast} p)\}\ [e] := e'\ \{p\}.}$$

One can derive (MUG) from (MUL) by using the frame rule:

$$\begin{array}{l} \{(e \mapsto -) \ast r\} \\ \quad \left.\begin{array}{l} \{e \mapsto -\} \\ [e] := e' \\ \{e \mapsto e'\} \end{array}\right\} \ast r \\ \{(e \mapsto e') \ast r\}, \end{array}$$

while to go in the opposite direction it is only necessary to take r to be **emp**:

$$\begin{array}{l} \{e \mapsto -\} \\ \{(e \mapsto -) \ast \mathbf{emp}\} \\ [e] := e' \\ \{(e \mapsto e') \ast \mathbf{emp}\} \\ \{e \mapsto e'\}. \end{array}$$

To derive (MUBR) from (MUG), we take r to be $(e \mapsto e') \mathrel{-\!\!\ast} p$ and use the derived axiom schema (2.5): $q \ast (q \mathrel{-\!\!\ast} p) \Rightarrow p$.

$$\begin{array}{l} \{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!\ast} p)\} \\ [e] := e' \\ \{(e \mapsto e') \ast ((e \mapsto e') \mathrel{-\!\!\ast} p)\} \\ \{p\}, \end{array}$$

while to go in the opposite direction we take p to be $(e \mapsto e') \ast r$ and use the derived axiom schema (2.7): $(p \ast r) \Rightarrow (p \ast (q \mathrel{-\!\!\ast} (q \ast r)))$.

$$\begin{array}{l} \{(e \mapsto -) \ast r\} \\ \{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!\ast} ((e \mapsto e') \ast r))\} \\ [e] := e' \\ \{(e \mapsto e') \ast r\}. \end{array}$$

For deallocation, there are only two rules, since the global form is also suitable for backward reasoning:

- The local form (DISL)

$$\overline{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}.}$$

- The global (and backward-reasoning) form (DISG)

$$\overline{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}.}$$

As with the mutation rules, one can derive (DISG) from (DISL) by using the frame rule, and go in the opposite direction by taking r to be **emp**.

3.8 Rules for Allocation

When we turn to the inference rules for allocation and lookup, our story becomes more complicated, since these commands modify variables. More precisely, they are what we will call *generalized assignment commands*, i.e., commands that first perform a computation that does not alter the store (though it may affect the heap), and then after this computation has finished, change the value of the store for a single variable.

However, neither allocation nor lookup are assignment commands in the sense we will use in this book, since they do not obey the rule (AS) for assignment. For example, if we tried to apply this rule to an allocation command, we could obtain

$$\{\mathbf{cons}(1,2) = \mathbf{cons}(1,2)\}\ \mathsf{x} := \mathbf{cons}(1,2)\ \{\mathsf{x} = \mathsf{x}\}, \qquad \text{(syntactically illegal)}$$

where the precondition is not a syntactically well-formed assertion, since $\mathbf{cons}(1,2)$ is not an expression — for the compelling reason that it has a side effect.

Even in the analogous case for lookup,

$$\{[\mathsf{y}] = [\mathsf{y}]\}\ \mathsf{x} := [\mathsf{y}]\ \{\mathsf{x} = \mathsf{x}\} \qquad \text{(syntactically illegal)}$$

is prohibited since $[\mathsf{y}]$ can have the side effect of aborting.

For allocation (and lookup) it is simplest to begin with local and global rules for the *nonoverwriting* case, where the old value of the variable being modified plays no role. For brevity, we abbreviate the sequence $e_1, \ldots, e_n$ of expressions by $\overline{e}$:

- The local nonoverwriting form (CONSNOL)

$$\overline{\{\textbf{emp}\}\ v := \textbf{cons}(\overline{e})\ \{v \mapsto \overline{e}\},}$$

  where $v \notin \mathrm{FV}(\overline{e})$.

- The global nonoverwriting form (CONSNOG)

$$\overline{\{r\}\ v := \textbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\},}$$

  where $v \notin \mathrm{FV}(\overline{e}, r)$.

As with mutation and deallocation, one can derive the global form from the local by using the frame rule, and the local from the global by taking r to be **emp**.

The price for the simplicity of the above rules is the prohibition of overwriting, which is expressed by the conditions $v \notin \mathrm{FV}(\overline{e})$ and $v \notin \mathrm{FV}(\overline{e}, r)$. Turning to the more complex and general rules that permit overwriting, we have three forms for allocation:

- The local form (CONSL)

$$\overline{\{v = v' \wedge \textbf{emp}\}\ v := \textbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\},}$$

  where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$ (i.e., each e'_i denotes $e_i/v \to v'$).

- The global form (CONSG)

$$\overline{\{r\}\ v := \textbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\},}$$

  where v' is distinct from v, $v' \notin \mathrm{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

- The backward-reasoning form (CONSBR)

$$\overline{\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\ v := \textbf{cons}(\overline{e})\ \{p\},}$$

  where v'' is distinct from v, $v'' \notin \mathrm{FV}(\overline{e}, p)$, and p'' denotes $p/v \to v''$.

To explain these rules, we begin with (CONSG). Here the existentially quantified variable v' denotes the old value of v, which has been overwritten by the allocation command (and may possibly no longer be determined by the store), much as in the alternative assignment rule (ASalt) in Section 3.2. A typical instance is

$$\{\mathsf{list}\ \alpha\ \mathsf{i}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}\}.$$

One can derive (CONSG) from the nonoverwriting rule (CONSNOG) by using a plausible equivalence that captures the essence of generalized assignment:

$$v := \mathbf{cons}(\overline{e}) \cong \mathbf{newvar}\ \hat{v}\ \mathbf{in}\ (\hat{v} := \mathbf{cons}(\overline{e})\ ;\ v := \hat{v}), \tag{3.1}$$

where $\hat{v}$ does not occur in $\overline{e}$ — and can be chosen not to occur in other specified phrases. (Here $\cong$ denotes an equivalence of meaning between two commands.)

We can regard this equivalence as defining the possibly overwriting case of allocation on the left by the nonoverwriting case on the right. Then we can derive (CONSG) from (CONSNOG) by existentially quantifying v and renaming it to v' (using the last axiom schema displayed in 2.2 in Section 2.2: $(p/v \to e) \Rightarrow (\exists v.\ p)$).

$$
\begin{array}{l}
\{r\} \\
\mathbf{newvar}\ \hat{v}\ \mathbf{in} \\
\quad \big(\hat{v} := \mathbf{cons}(\overline{e})\ ; \\
\quad \{(\hat{v} \mapsto \overline{e})\ *\ r\} \\
\quad \{\exists v.\ (\hat{v} \mapsto \overline{e})\ *\ r\} \\
\quad \{\exists v'.\ (\hat{v} \mapsto \overline{e}')\ *\ r'\} \\
\quad v := \hat{v}\big) \\
\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}
\end{array}
$$

One might expect the local rule to be

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\}},$$

(where v' is distinct from v and $v' \notin \mathrm{FV}(\overline{e})$), which can be derived from (CONSG) by taking r to be $\mathbf{emp}$. But this rule, though sound, is too weak.

For example, the postcondition of the instance

$$\{\mathbf{emp}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\}$$

gives no information about the second component of the new record.

In the stronger local rule (CONSL), the existential quantifier is dropped and v' becomes a *ghost variable* — a device that is used commonly in Hoare logic to relate values in different states of the computation.

A *ghost variable* in a specification $\{p\}\ c\ \{q\}$ is a variable that occurs in p or q but is not modified by c. (Often, the ghost variable is not permitted to occur free in c at all.) Since the ghost is not modified, it is evident that it denotes the same value in the pre- and postcondition.

For example, the following instance of (CONSL)

$$\{\mathsf{i} = \mathsf{j} \wedge \mathbf{emp}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\mathsf{i} \mapsto 3, \mathsf{j}\}$$

shows that the value of j in the postcondition is value of i before the assignment.

We can derive (CONSL) from (CONSG) by replacing r by $v = v' \wedge \mathbf{emp}$ and v' by v'', and using the fact that $v'' = v'$ is pure, plus simple properties of equality and the existential quantifier:

$$\begin{array}{l}
\{v = v' \wedge \mathbf{emp}\} \\
v := \mathbf{cons}(\overline{e}) \\
\{\exists v''.\ (v \mapsto \overline{e}'')\ *\ (v'' = v' \wedge \mathbf{emp})\} \\
\{\exists v''.\ ((v \mapsto \overline{e}'') \wedge v'' = v')\ *\ \mathbf{emp}\} \\
\{\exists v''.\ (v \mapsto \overline{e}'') \wedge v'' = v'\} \\
\{\exists v''.\ (v \mapsto \overline{e}') \wedge v'' = v'\} \\
\{v \mapsto \overline{e}'\}.
\end{array}$$

(Here v'' is chosen to be distinct from v, v', and the free variables of $\overline{e}$.)

Then to complete the circle (and show the adequacy of (CONSL)), we derive (CONSG) from (CONSL) by using the frame rule and (EQ):

$$
\begin{array}{l}
\{r\} \\
\{\exists v'.\ v = v' \wedge r\} \\
\quad \left.\begin{array}{l}
\{v = v' \wedge r\} \\
\{v = v' \wedge r'\} \\
\{v = v' \wedge (\mathbf{emp}\ *\ r')\} \\
\{(v = v' \wedge \mathbf{emp})\ *\ r'\} \\
\quad \left.\begin{array}{l}
\{(v = v' \wedge \mathbf{emp})\} \\
v := \mathbf{cons}(\overline{e}) \\
\{(v \mapsto \overline{e}')\}
\end{array}\right\}\ *\ r' \\
\{(v \mapsto \overline{e}')\ *\ r'\}
\end{array}\right\}\ \exists v' \\
\{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}
\end{array}
$$

In the backward-reasoning rule (CONSBR), the universal quantifier $\forall v''$ in the precondition expresses the nondeterminacy of allocation. (We have chosen the metavariable v'' rather than v' to simplify some deriviations.) The most direct way to see this is by a semantic proof that the rule is sound:

Suppose the precondition holds in the state s, h, i.e., that

$$s, h \models \forall v''.\ (v'' \mapsto \overline{e}) -\!\!* \ p''.$$

Then the semantics of universal quantification gives

$$\forall \ell.\ [\, s \mid v''\colon \ell\,], h \models (v'' \mapsto \overline{e}) -\!\!* \ p'',$$

and the semantics of separating implication gives

$$\forall \ell, h'.\ h \perp h' \text{ and } \underline{[\, s \mid v''\colon \ell\,], h' \models (v'' \mapsto \overline{e})} \text{ implies } [\, s \mid v''\colon \ell\,], h \cdot h' \models p'',$$

where the underlined formula is equivalent to

$$h' = [\, \ell\colon [\![e_1]\!]_{\exp} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\exp} s\,].$$

Thus

$$
\begin{array}{l}
\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h \text{ implies} \\
\qquad [\, s \mid v''\colon \ell\,], [\, h \mid \ell\colon [\![e_1]\!]_{\exp} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\exp} s\,] \models p''\big).
\end{array}
$$

Then, by Proposition 3 in Chapter 2, since p'' denotes $p/v \to v''$, we have $[\, s \mid v''\colon \ell\,], h' \models p''$ iff $\hat{s}, h' \models p$, where

$$\hat{s} = [\, s \mid v''\colon \ell \mid v\colon [\![v'']\!]_{\exp}[\, s \mid v''\colon \ell\,]\,] = [\, s \mid v''\colon \ell \mid v\colon \ell\,].$$

Moreover, since v'' does not occur free in p, we can simplify $\hat{s}, h' \models p$ to $[\, s \mid v\colon \ell\,], h' \models p$. Thus

$$\begin{aligned}&\forall \ell.\ \big(\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h \text{ implies}\\ &\qquad [\, s \mid v\colon \ell\,], [\, h \mid \ell\colon [\![e_1]\!]_{\exp} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\exp} s\,] \models p\big).\end{aligned}$$

Now execution of the allocation command $v := \mathbf{cons}(\overline{e})$, starting in the state s, h, will never abort, and will always termininate in a state $[\, s \mid v\colon \ell\,], [\, h \mid \ell\colon [\![e_1]\!]_{\exp} s \mid \ldots \mid \ell + n - 1\colon [\![e_n]\!]_{\exp} s\,]$ for some ℓ such that $\ell, \ldots, \ell + n - 1 \notin \operatorname{dom} h$. Thus the condition displayed above insures that all possible terminating states satisfy the postcondition p.

We also show that (CONSBR) and (CONSG) are interderivable. To derive (CONSBR) from (CONSG), we choose $v' \notin \mathrm{FV}(\overline{e}, p)$ to be distinct from v and v'', take r to be $\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''$, and use predicate-calculus properties of quantifiers, as well as (2.5): $q * (q \mathrel{-\!\!*} p) \Rightarrow p$.

$$\begin{aligned}&\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p''\}\\ &v := \mathbf{cons}(\overline{e})\\ &\{\exists v'.\ (v \mapsto \overline{e}') * (\forall v''.\ (v'' \mapsto \overline{e}') \mathrel{-\!\!*} p'')\}\\ &\{\exists v'.\ (v \mapsto \overline{e}') * ((v \mapsto \overline{e}') \mathrel{-\!\!*} p)\}\\ &\{\exists v'.\ p\}\\ &\{p\}.\end{aligned}$$

To go in the other direction, we choose $v'' \notin \mathrm{FV}(\overline{e}, r)$ to be distinct from v and v', take p to be $\exists v'.\ (v \mapsto \overline{e}') * r'$, and use properties of quantifiers, as well as (2.6): $r \Rightarrow (q \mathrel{-\!\!*} (q \ast r))$.

$$
\begin{aligned}
&\{r\}\\
&\{\forall v''.\ r\}\\
&\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} ((v'' \mapsto \overline{e}) \ast r)\}\\
&\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (((v'' \mapsto \overline{e}') \ast r')/v' \to v)\}\\
&\{\forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} (\exists v'.\ (v'' \mapsto \overline{e}') \ast r')\}\\
&v := \mathbf{cons}(\overline{e})\\
&\{\exists v'.\ (v \mapsto \overline{e}') \ast r'\}.
\end{aligned}
$$

3.9 Rules for Lookup

Finally, we come to the lookup command, which — for no obvious reason — has the richest variety of inference rules. We begin with the nonoverwriting rules:

- The local nonoverwriting form (LKNOL)

$$\frac{}{\{e \mapsto v''\}\ v := [e]\ \{v = v'' \wedge (e \mapsto v)\},}$$

  where $v \notin \mathrm{FV}(e)$.

- The global nonoverwriting form (LKNOG)

$$\frac{}{\{\exists v''.\ (e \mapsto v'') \ast p''\}\ v := [e]\ \{(e \mapsto v) \ast p\},}$$

  where $v \notin \mathrm{FV}(e)$, $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

In (LKNOG), there is no restriction preventing v'' from being the same variable as v. Thus, as a special case,

$$\frac{}{\{\exists v.\ (e \mapsto v) \ast p\}\ v := [e]\ \{(e \mapsto v) \ast p\},}$$

where $v \notin \mathrm{FV}(e)$. For example, if we take

$$
\begin{array}{lll}
v & \text{to be} & \mathsf{j}\\
e & \text{to be} & \mathsf{i}+1
\end{array}
\qquad p \ \text{ to be } \ \mathsf{i} \mapsto 3 \ast \mathbf{list}\,\alpha\,\mathsf{j},
$$

(and remember that $\mathsf{i} \mapsto 3, \mathsf{j}$ abbreviates $(\mathsf{i} \mapsto 3) * (\mathsf{i}+1 \mapsto \mathsf{j})$), then we obtain the instance

$$\begin{aligned}
&\{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j} * \textbf{list}\ \alpha\ \mathsf{j}\}\\
&\mathsf{j} := [\mathsf{i}+1]\\
&\{\mathsf{i} \mapsto 3, \mathsf{j} * \textbf{list}\ \alpha\ \mathsf{j}\}.
\end{aligned}$$

In effect, the action of the lookup command is to erase an existential quantifier. In practice, if one chooses the names of quantified variables with foresight, most lookup commands can be described by this simple special case.

Turning to the rules for the general case of lookup, we have

- The local form (LKL)

  $$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

  where v, v', and v'' are distinct, and e' denotes $e/v \to v'$.

- The global form (LKG)

  $$\frac{}{\begin{array}{l}\{\exists v''.\ (e \mapsto v'') * (r/v' \to v)\}\ v := [e]\\ \qquad\qquad\qquad\qquad \{\exists v'.\ (e' \mapsto v) * (r/v'' \to v)\},\end{array}}$$

  where v, v', and v'' are distinct, v', $v'' \notin \mathrm{FV}(e)$, $v \notin \mathrm{FV}(r)$, and e' denotes $e/v \to v'$.

- The first backward-reasoning form (LKBR1)

  $$\frac{}{\{\exists v''.\ (e \mapsto v'') * ((e \mapsto v'') \mathrel{-\!\!*} p'')\}\ v := [e]\ \{p\},}$$

  where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

- The second backward-reasoning form (LKBR2)

  $$\frac{}{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\},}$$

  where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

In each of these rules, one can think of v' as denoting the value of v before execution of the lookup command, and v'' as denoting the value of v after execution.

We begin with a semantic proof of the soundness of the local rule (LKL). Suppose that the precondition holds in the state s_0, h, i.e., that

$$s_0, h \models v = v' \wedge (e \mapsto v'').$$

Then $s_0\, v = s_0\, v'$ and $h = [\, [\![e]\!]_{\mathrm{exp}} s_0 \colon s_0\, v''\,]$.

Starting in the state s_0, h, the execution of $v := [e]$ will not abort (since $[\![e]\!]_{\mathrm{exp}} s_0 \in \mathrm{dom}\, h$), and will terminate with the store $s_1 = [\, s_0 \mid v \colon s_0\, v''\,]$ and the unchanged heap h. To see that this state satisfies the postcondition, we note that $s_1\, v = s_0\, v'' = s_1\, v''$ and, since e' does not contain v, $[\![e']\!]_{\mathrm{exp}} s_1 = [\![e']\!]_{\mathrm{exp}} s_0$. Then by applying Proposition 3 in Chapter 2, with $\hat{s} = [\, s_0 \mid v \colon s_0\, v'\,] = [\, s_0 \mid v \colon s_0\, v\,] = s_0$, we obtain $[\![e']\!]_{\mathrm{exp}} s_0 = [\![e]\!]_{\mathrm{exp}} s_0$. Thus $h = [\, [\![e']\!]_{\mathrm{exp}} s_1 \colon s_1\, v\,]$, and $s_1, h \models v = v'' \wedge (e' \mapsto v)$.

To derive (LKG) from (LKL), we use the frame rule and two applications of (EQ):

$$
\begin{array}{l}
\{\exists v''.\ (e \mapsto v'') \ast (r/v' \to v)\}\\
\{\exists v', v''.\ (v = v' \wedge (e \mapsto v'')) \ast (r/v' \to v)\}\\
\quad \left.\begin{array}{l}
\{\exists v''.\ (v = v' \wedge (e \mapsto v'')) \ast (r/v' \to v)\}\\
\quad \left.\begin{array}{l}
\{(v = v' \wedge (e \mapsto v'')) \ast (r/v' \to v)\}\\
\quad \left.\begin{array}{l}
\{v = v' \wedge (e \mapsto v'')\}\\
v := [e]\\
\{v = v'' \wedge (e' \mapsto v)\}
\end{array}\right\} \ast\ r\\
\{(v = v'' \wedge (e' \mapsto v)) \ast (r/v'' \to v)\}
\end{array}\right\} \exists v''\\
\{\exists v''.\ (v = v'' \wedge (e' \mapsto v)) \ast (r/v'' \to v)\}
\end{array}\right\} \exists v'\\
\{\exists v', v''.\ (v = v'' \wedge (e' \mapsto v)) \ast (r/v'' \to v)\}\\
\{\exists v'.\ (e' \mapsto v) \ast (r/v'' \to v)\}.
\end{array}
$$

The global form (LKG) is the most commonly used of the rules that allow overwriting As an example of an instance, if we take

$$
\begin{array}{rll}
v & \text{to be} & \mathsf{j}\\
v' & \text{to be} & \mathsf{m}\\
v'' & \text{to be} & \mathsf{k}
\end{array}
\qquad
\begin{array}{rll}
e & \text{to be} & \mathsf{j}+1\\
r & \text{to be} & \mathsf{i}+1 \mapsto \mathsf{m} \ast \mathsf{k}+1 \mapsto \mathbf{nil},
\end{array}
$$

then we obtain (with a little use of the commutivity of $*$)

$$\begin{aligned}
&\{\exists \mathsf{k}.\ \mathsf{i}+1 \mapsto \mathsf{j} \ * \ \mathsf{j}+1 \mapsto \mathsf{k} \ * \ \mathsf{k}+1 \mapsto \mathbf{nil}\} \\
&\mathsf{j} := [\mathsf{j}+1] \\
&\{\exists \mathsf{m}.\ \mathsf{i}+1 \mapsto \mathsf{m} \ * \ \mathsf{m}+1 \mapsto \mathsf{j} \ * \ \mathsf{j}+1 \mapsto \mathbf{nil}\}.
\end{aligned}$$

To derive (LKL) from (LKG), we first rename the variables v' and v'' in (LKG) to be $\hat{v}'$ and $\hat{v}''$, chosen not to occur free in e, and then replace r in (LKG) by $\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp}$. Then we use the purity of equality, and predicate-calculus properties of equality and the existential quantifier, to obtain

$$\begin{aligned}
&\{v = v' \wedge (e \mapsto v'')\} \\
&\{\exists \hat{v}''.\ v = v' \wedge \hat{v}'' = v'' \wedge (e \mapsto \hat{v}'')\} \\
&\{\exists \hat{v}''.\ (e \mapsto \hat{v}'') \ * \ (v = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})\} \\
&\{\exists \hat{v}''.\ (e \mapsto \hat{v}'') \ * \ ((\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})/\hat{v}' \to v)\} \\
&v := [e] \\
&\{\exists \hat{v}'.\ (\hat{e}' \mapsto v) \ * \ ((\hat{v}' = v' \wedge \hat{v}'' = v'' \wedge \mathbf{emp})/\hat{v}'' \to v)\} \\
&\{\exists \hat{v}'.\ (\hat{e}' \mapsto v) \ * \ (\hat{v}' = v' \wedge v = v'' \wedge \mathbf{emp})\} \\
&\{\exists \hat{v}'.\ \hat{v}' = v' \wedge v = v'' \wedge (\hat{e}' \mapsto v)\} \\
&\{v = v'' \wedge (e' \mapsto v)\}
\end{aligned}$$

(where $\hat{e}'$ denotes $e/v \to \hat{v}'$).

Turning to the backward-reasoning rules, to derive (LKBR1) from (LKG), we take r to be $(e' \mapsto v'') -\!\!* \ p''$ and use the axiom schema (2.5): $q \ * \ (q -\!\!* \ p) \Rightarrow p$. (In the first line, we can rename the quantified variable v'' to be any variable not in $\mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$.)

$$\begin{aligned}
&\{\exists v''.\ (e \mapsto v'') \ * \ ((e \mapsto v'') -\!\!* \ p'')\} \\
&v := [e] \\
&\{\exists v'.\ (e' \mapsto v) \ * \ ((e' \mapsto v) -\!\!* \ p)\} \\
&\{\exists v'.\ p\} \\
&\{p\},
\end{aligned}$$

and to derive (LKBR2) from (LKBR1), we use the last axiom schema in (2.4): $(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') * ((e \mapsto e') -\!\!* \, p)$.

$$\begin{aligned}
&\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\\
&\{\exists v''.\ (e \mapsto v'') * ((e \mapsto v'') -\!\!* \, p'')\}\\
&v := [e]\\
&\{p\}.
\end{aligned}$$

Then to derive (LKL) from (LKBR2), we rename v'' to $\hat{v}$ in the precondition of (LKBR2), take p to be $v = v'' \wedge (e' \mapsto v)$, and use properties of $\hookrightarrow$, equality, and the existential quantifier:

$$\begin{aligned}
&\{v = v' \wedge (e \mapsto v'')\}\\
&\{v = v' \wedge (e \hookrightarrow v'') \wedge (e' \mapsto v'')\}\\
&\{(e \hookrightarrow v'') \wedge (e' \mapsto v'')\}\\
&\{\exists \hat{v}.\ (e \hookrightarrow v'') \wedge \hat{v} = v'' \wedge (e' \mapsto v'')\}\\
&\{\exists \hat{v}.\ (e \hookrightarrow \hat{v}) \wedge \hat{v} = v'' \wedge (e' \mapsto \hat{v})\}\\
&v := [e]\\
&\{v = v'' \wedge (e' \mapsto v)\}.
\end{aligned}$$

The reader may verify that we have given more than enough derivations to establish that all of the rules for lookup that permit overwriting are interderivable.

For good measure, we also derive the global nonoverwriting rule (LKNOG) from (LKG): Suppose v and v'' are distinct variables, $v \notin \mathrm{FV}(e)$, and $v'' \notin \mathrm{FV}(e) \cup \mathrm{FV}(p)$. We take v' to be a variable distinct from v and v'' that does not occur free in e or p, and r to be $p'' = p/v \to v''$. (Note that $v \notin \mathrm{FV}(e)$ implies that $e' = e$.) Then

$$\begin{aligned}
&\{\exists v''.\ (e \mapsto v'') * p''\}\\
&\{\exists v''.\ (e \mapsto v'') * (p''/v' \to v)\}\\
&v := [e]\\
&\{\exists v'.\ (e' \mapsto v) * (p''/v'' \to v)\}\\
&\{\exists v'.\ (e \mapsto v) * p)\}\\
&\{(e \mapsto v) * p)\}.
\end{aligned}$$

In the first line, we can rename the quantified variable v'' to be any variable not in $\mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, so that v'' does not need to be distinct from v.

3.10 Weakest Preconditions for the New Commands

The *weakest precondition* of a command c for an assertion q, written $\mathbf{wp}(c, q)$, is the weakest assertion p such that the total correctness specification $[\,p\,]\ c\ [\,q\,]$ is valid. In other words, $[\,\mathbf{wp}(c, q)\,]\ c\ [\,q\,]$ is valid, and, for any p such that $[\,p\,]\ c\ [\,q\,]$, $p \Rightarrow \mathbf{wp}(c, q)$ is valid.

Similarly, the *weakest liberal precondition* of a command c for an assertion q, written $\mathbf{wlp}(c, q)$, is the weakest assertion p such that the partial correctness specification $\{p\}\ c\ \{q\}$ is valid.

Although we will not use weakest preconditions [41, 42] extensively in this book, it should be noted that, for each of the new heap-manipulating commands, the backward-reasoning form gives the weakest liberal precondition:

- Mutation

$$\mathbf{wlp}([e] := e', p) = (e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!*} p).$$

- Allocation

$$\mathbf{wlp}(v := \mathbf{cons}(\overline{e}), p) = \forall v''.\ (v'' \mapsto \overline{e}) \mathrel{-\!\!*} p'',$$

  where v'' is distinct from v, $v'' \notin \mathrm{FV}(\overline{e}, p)$, and p'' denotes $p/v \to v''$.

- Disposal

$$\mathbf{wlp}(\mathbf{dispose}\ e, r) = (e \mapsto -) \ast r.$$

- Lookup

$$\mathbf{wlp}(v := [e], p) = \exists v''.\ (e \hookrightarrow v'') \wedge p'',$$

  where $v'' \notin \mathrm{FV}(e) \cup (\mathrm{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

Moreover, since these commands always terminate, the same equations hold for the weakest precondition $\mathbf{wp}$.

3.11 A Final Example

In conclusion, we reprise the annotated specification given at the end of Section 1.5, this time indicating the particular inference rules and verification conditions that are used. (To make the action of the inference rules clear, we have also given the unabbreviated form of each assertion.)

$$\{\mathbf{emp}\}$$

$$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a})\ ; \qquad \text{(CONSNOL)}$$

$$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\} \quad \text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{a}\}$$

$$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b})\ ; \qquad \text{(CONSNOG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{a} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$$

$$(p/v \to e \Rightarrow \exists v.\ p)$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, -) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast (\exists \mathsf{a}.\ \mathsf{x} + 1 \mapsto \mathsf{a}) \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$$

$$[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x}\ ; \qquad \text{(MUG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$$

$$(p/v \to e \Rightarrow \exists v.\ p)$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast (\exists \mathsf{b}.\ \mathsf{y} + 1 \mapsto \mathsf{b})\}$$

$$[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y}\ ; \qquad \text{(MUG)}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{x} - \mathsf{y}\}$$

$$(p/v \to e \Rightarrow \exists v.\ p)$$

$$\{\exists \mathsf{o}.\ (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \ast (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b},\ -\mathsf{o})\}$$

$$\text{i.e., } \{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{o} \ast \mathsf{x} + \mathsf{o} \mapsto \mathsf{b} \ast \mathsf{x} + \mathsf{o} + 1 \mapsto -\mathsf{o}\}$$

Exercise 3

Fill in the postconditions in

$$\{(e_1 \mapsto -) \ast (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}$$

$$\{(e_1 \mapsto -) \land (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}.$$

to give two sound inference rules describing a sequence of two mutations. Your postconditions should be as strong as possible.

Give a derivation of each of these inference rules, exhibited as an annotated specification.

Exercise 4

The following are alternative global rules for allocation and lookup that use ghost variables (v' and v''):

- The ghost-variable global form for allocation (CONSGG)

$$\frac{}{\{v = v' \land r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e}') \ast r'\},}$$

  where v' is distinct from v, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

- The ghost-variable global form for lookup (LKGG)

$$\frac{}{\{v = v' \land ((e \mapsto v'') \ast r)\}\ v := [e]\ \{v = v'' \land ((e' \mapsto v) \ast r)\},}$$

  where v, v', and v'' are distinct, $v \notin \mathrm{FV}(r)$, and e' denotes $e/v \to v'$.

Derive (CONSGG) from (CONSG), and (CONSL) from (CONSGG). Derive (LKGG) from (LKG), and (LKL) from (LKGG).

Exercise 5

Derive (LKNOL) from (LKNOG) and vice-versa. (Hint: To derive (LKNOL) from (LKNOG), use the version of (LKNOG) where $v'' = v$. To derive (LKNOG) from (LKNOL), assume v and v'' are distinct, and then apply renaming of v'' in the precondition to cover the case where $v = v''$.)

Exercise 6

Closely akin to (3.1) is the following equivalence of meaning between two lookup commands:

$$v := [e] \cong \textbf{newvar}\ \hat{v}\ \textbf{in}\ (\hat{v} := [e]\ ;\ v := \hat{v}). \tag{3.2}$$

Use this equivalence to derive (LKG) from (LKNOG).