Chapter 3

Hoare Logic [Lecture 5–6]

Version of February 18, 2024

Summary We present the formal system of Hoare Logic, with particular emphasis on linearized proof presentations. We also show soundness of the logic with respect to the big-step operational semantics of IMP, and discuss some consequences of its completeness.

3.1 Specifications and proofs

3.1.1 Introduction

Consider the following simple IMP program:

$$SUM \;\equiv\; (s := 0; i := 0; \textbf{while } i \neq n \textbf{ do } (i := i + 1; s := s + i))$$

(For readability, we have elided the overbars over all numerals, and written $a_0 \neq a_1$ for $\neg(a_0 = a_1)$.) Suppose $\sigma(n) = 100$, and $\langle SUM, \sigma\rangle \downarrow \sigma'$. What is $\sigma'(s)$? It is *in principle* fairly straightforward to mechanically construct the postulated execution derivation, from which we can immediately read off that $\sigma'(s) = 5050$. However, needless to say, actually constructing the derivation by hand would be a massive, and largely pointless, exercise.

An alternative, and much more satisfying, approach would be to convince ourselves that the program is actually computing, in s, the value of $\sum_{i=1}^{n} i$, and use – like in the anecdote about the young Carl Gauss – the formula for arithmetic series to calculate that $s = n \times (n+1)/2 = 5050$. However, while the correctness of the summation formula itself can be quickly established by simple mathematical induction, the task of establishing that the program is actually computing precisely that sum (and not, say, an off-by-one variant), presumably by induction over derivations using the formal operational-semantics rules, would still be quite tedious.

In this chapter, we will consider a third approach that will allow us to prove properties of programs like the above much more smoothly and concisely. We will introduce a *program logic* for showing *directly* that an IMP program satisfies a specification, abstracting from the operational semantics of the various language constructs into a collection of proof rules about them. Such a program logic is sometimes called an *axiomatic semantics* of the language.

3.1.2 Operational semantics of commands

The definition of IMP's big-step command execution judgment $\boxed{\langle c, \sigma\rangle \downarrow \sigma'}$ from Figure 2.1 is included here again, for easy reference:

$$\text{EC-Skip}: \frac{}{\langle \textbf{skip}, \sigma\rangle \downarrow \sigma} \quad \text{EC-Assign}: \frac{\langle a, \sigma\rangle \downarrow n}{\langle X := a, \sigma\rangle \downarrow \sigma[X \mapsto n]} \quad \text{EC-Seq}: \frac{\langle c_0, \sigma\rangle \downarrow \sigma'' \quad \langle c_1, \sigma''\rangle \downarrow \sigma'}{\langle c_0; c_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{EC-IfT}: \frac{\langle b, \sigma\rangle \downarrow \textbf{true} \quad \langle c_0, \sigma\rangle \downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \downarrow \sigma'} \quad \text{EC-IfF}: \frac{\langle b, \sigma\rangle \downarrow \textbf{false} \quad \langle c_1, \sigma\rangle \downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{EC-WhileF}: \frac{\langle b, \sigma\rangle \downarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } c_0, \sigma\rangle \downarrow \sigma} \quad \text{EC-WhileT}: \frac{\langle b, \sigma\rangle \downarrow \textbf{true} \quad \langle c_0, \sigma\rangle \downarrow \sigma'' \quad \langle \textbf{while } b \textbf{ do } c_0, \sigma''\rangle \downarrow \sigma'}{\langle \textbf{while } b \textbf{ do } c_0, \sigma\rangle \downarrow \sigma'}$$

3.1.3 Assertions

An *assertion* is a logical formula saying something about the (expected) relationships between the values of store locations at a particular program point. The syntax of assertions is as follows:

$$A ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \le a_1 \mid \neg A_0 \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid A_0 \Rightarrow A_1 \mid \forall X. A_0 \mid \exists X. A_0$$

(We will frequently also use other uppercase Latin letters, though not X, Y, or Z, for assertions.) Unlike *FSoPL* (and some other texts), we will not introduce a separate sort of "integer variables" for use in quantified assertions, but reuse the existing notion of locations for that purpose. This simplification will not cause any significant issues in our limited language.

Note that every boolean expression b is also a well formed assertion A, but the converse is not true, because assertions may contain implications and quantifiers. If we don't care about the distinction between classical and constructive proofs, we can simply treat $A_0 \Rightarrow A_1$ as an abbreviation for $\neg A_0 \vee A_1$; but the addition of quantifiers makes an essential difference. For example, we may express that location x contains an even number by the assertion $\exists i.\, x = 2 \times i$, even though we cannot use this same formula as a condition in an **if**-command to *test* for evenness.

With quantifiers potentially present, we also cannot express the meaning of assertions in terms of an operational semantics, but we can still define a *satisfaction relation* $\sigma \models A$, by induction on the syntactic structure of A:

$$\begin{aligned}
\sigma &\models \textbf{true} &&\iff \top \quad \text{(i.e., } \sigma \models \textbf{true} \text{ holds for every } \sigma\colon \Sigma\text{)} \\
\sigma &\models \textbf{false} &&\iff \bot \quad \text{(i.e., } \sigma \models \textbf{false} \text{ holds for no } \sigma\colon \Sigma\text{)} \\
\sigma &\models a_0 = a_1 &&\iff \exists n\colon \mathbb{Z}.\, \langle a_0, \sigma\rangle \downarrow n \wedge \langle a_1, \sigma\rangle \downarrow n \\
\sigma &\models a_0 \le a_1 &&\iff \exists n_0, n_1\colon \mathbb{Z}.\, \langle a_0, \sigma\rangle \downarrow n_0 \wedge \langle a_1, \sigma\rangle \downarrow n_1 \wedge n_0 \le n_1 \\
\sigma &\models \neg A_0 &&\iff \neg(\sigma \models A_0), \quad \text{often written as } \sigma \not\models A \\
\sigma &\models A_0 \wedge A_1 &&\iff (\sigma \models A_0) \wedge (\sigma \models A_1) \\
\sigma &\models A_0 \vee A_1 &&\iff (\sigma \models A_0) \vee (\sigma \models A_1) \\
\sigma &\models A_0 \Rightarrow A_1 &&\iff (\sigma \models A_0) \Rightarrow (\sigma \models A_1) \\
\sigma &\models \forall X. A_0 &&\iff \forall n\colon \mathbb{Z}.\, (\sigma[X \mapsto n] \models A_0) \\
\sigma &\models \exists X. A_0 &&\iff \exists n\colon \mathbb{Z}.\, (\sigma[X \mapsto n] \models A_0)
\end{aligned}$$

Note how the meaning of each kind of assertion formula is given in terms of the meanings of its constituent subformula(s), if any.

The common syntax of assertions and boolean expressions is more than a coincidence. Their semantics also agree, in the following sense:

Lemma 3.1 (Conservative extension) *For any $b \in$ **Bexp**, we have $(\sigma \models b) \Leftrightarrow \langle b, \sigma\rangle \downarrow$ **true**.*

Proof. By induction on the structure of b. Note that, for each possible form of b, we prove both directions of the biimplication together (but still separately), rather than first proving ($\Rightarrow$) for all b and then ($\Leftarrow$) for all b. Two illustrative cases are:

- Case $b = (a_0 = a_1)$. For the ($\Rightarrow$) direction, assume $\sigma \models a_0 = a_1$, i.e., there exists an n such that $\langle a_0, \sigma\rangle \downarrow n$ and $\langle a_1, \sigma\rangle \downarrow n$. Putting those derivations together using EB-EQT gives us the required derivation of $\langle a_0 = a_1, \sigma\rangle \downarrow$ **true**.

  Conversely, for ($\Leftarrow$), suppose $\langle a_0 = a_1, \sigma\rangle \downarrow$ **true**. That could only have been derived by the EB-EQT rule (because EB-EQF returns **false**, and the none of the other rules even talk about $a_0 = a_1$). Thus, we must have subderivations of $\langle a_0, \sigma\rangle \downarrow n$ and $\langle a_1, \sigma\rangle \downarrow n$ for some n, exactly as needed to show that $\sigma \models a_0 = a_1$.

- Case $b = \neg b_0$. This case is actually rather subtle. For the ($\Rightarrow$) direction, suppose $\sigma \models \neg b_0$, i.e., that $\sigma \models b_0$ does *not* hold. By the IH on b_0 (using the ($\Leftarrow$) direction of the biimplication!), we conclude that then $\langle b_0, \sigma\rangle \downarrow$ **true** cannot hold either (because if it did, that would imply $\sigma \models b_0$, and thus a contradiction). But since boolean evaluation is still total (Theorem 2.4), we must instead have $\langle b_0, \sigma\rangle \downarrow$ **false**, and thus, by EB-NEGF, $\langle \neg b_0, \sigma\rangle \downarrow$ **true**, as required.

  Conversely, for ($\Leftarrow$), suppose $\langle \neg b_0, \sigma\rangle \downarrow$ **true**. This could only have been derived using the EB-NEGF rule, so we must have that $\langle b_0, \sigma\rangle \downarrow$ **false**. But since boolean evaluation is deterministic (Theorem 2.6), this means that we cannot also have $\langle b_0, \sigma\rangle \downarrow$ **true**. And thus, by the IH on b_0 (using the ($\Rightarrow$) direction!), $\sigma \models b_0$ cannot hold either (because it would imply $\langle b_0, \sigma\rangle \downarrow$ **true**, a contradiction). But that says precisely that $\sigma \models \neg b_0$ does hold, again as required.

■

Definition 3.2 (Validity of assertions) *Validity of an assertion is defined as follows:*

$$\models A \iff \forall \sigma : \Sigma.\, (\sigma \models A)$$

We may think of valid assertions as the universally valid formulas of standard integer arithmetic, e.g., $\models \exists x.\, x \leq y$, or $\models \neg\exists x.\, \forall y.\, x \leq y$.

3.1.4 Hoare triples

A *Hoare triple* (or *partial correctness assertion*) $\{A\}\, c\, \{B\}$ consists of two assertions A and B (called the *precondition* and *postcondition*, respectively), and a command c. (Hoare triples are named for Sir Tony Hoare, who proposed them (and the associated logic) in 1969 as a concise formulation of a program satisfying a specification.)

Definition 3.3 (Validity of Hoare triples) *Validity of a Hoare triple, with respect to the operational semantics, is defined as follows:*

$$\models \{A\}\, c\, \{B\} \iff \forall \sigma, \sigma' : \Sigma.\, (\sigma \models A) \wedge \langle c, \sigma \rangle \downarrow \sigma' \Rightarrow (\sigma' \models B)$$

That is, the triple is valid precisely if, whenever the command is executed in a starting state satisfying the precondition, the final state (if any) satisfies the postcondition.

For example, we should be able to establish

$$\models \{\mathbf{true}\}\, SUM\, \{2 \times s = n \times (n+1)\}\,.$$

Here, since we don't have division available as an arithmetic operator in formulas, we have expressed $s = n \times (n+1) \,/\, 2$ in an equivalent way, with both sides of the equality multiplied by 2; this also makes it immediately evident that the division will not have a remainder.

More subtly, the above specification technically holds even when the value of n in the initial store is negative. This is because, as can be easily seen (at least informally), in that case the program will actually run forever, letting i run through the natural numbers, as the loop condition $i \neq n$ will always be true for $n < 0$. Thus, the assumption inherent in the specification, "if/when the program terminates, the final state will satisfy" is *vacuously* true. We will return briefly to the question of also guaranteeing termination in Section 3.1.6.

Like for ordinary logic, we also have a more syntactic, finitely verifiable notion of Hoare-triple correctness. (We will see shortly that these two notions are, in fact, equivalent.) This so-called *Hoare logic* can be presented very compactly:

Definition 3.4 (Provability of Hoare triples) *The judgment* $\boxed{\vdash \{A\}\, c\, \{B\}}$ *is defined by the following rules (omitting* $\vdash$ *everywhere for conciseness):*

$$\text{H-Skip} : \frac{}{\{A\}\, \mathbf{skip}\, \{A\}} \qquad \text{H-Assign} : \frac{}{\{B[a/X]\}\, X := a\, \{B\}} \qquad \text{H-Seq} : \frac{\{A\}\, c_0\, \{C\} \quad \{C\}\, c_1\, \{B\}}{\{A\}\, c_0; c_1\, \{B\}}$$

$$\text{H-If} : \frac{\{A \wedge b\}\, c_0\, \{B\} \quad \{A \wedge \neg b\}\, c_1\, \{B\}}{\{A\}\, \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\, \{B\}} \qquad \text{H-While} : \frac{\{A \wedge b\}\, c_0\, \{A\}}{\{A\}\, \mathbf{while}\ b\ \mathbf{do}\ c_0\, \{A \wedge \neg b\}}$$

$$\text{H-Conseq} : \frac{\models A \Rightarrow A' \qquad \{A'\}\, c\, \{B'\} \qquad \models B' \Rightarrow B}{\{A\}\, c\, \{B\}}$$

In the *consequence rule* (H-Conseq), the first and last premise would be more properly written as side conditions (since they represent semantic criteria, rather than subderivations), but it is somewhat conventional to write them above the line.

When constructing a non-trivial derivation in Hoare logic, by far the hardest task is coming up with suitable *loop invariants* A for all uses of the H-While rule. Much like an induction hypothesis, the invariant needs to be *general* enough to encompass all possible states encountered when (re)entering the loop, but also *specific* enough to – in conjunction with the negated loop condition b – ensure that the desired postcondition holds after the loop.

For our summation program, finding the invariant as actually fairly immediate. We observe that, whenever we are at the top of the loop (just before evaluating the loop

test), the relationship $2 \times s = i \times (i+1)$ holds: it is trivially true before the first iteration, when $s = i = 0$, and we will see in a moment that it also holds again immediately after each execution of the loop body. This means that, just after we exit the loop, we also have $2 \times s = i \times (i+1)$, but now also $i = n$ (because the loop condition must have just evaluated to **false**). But those two equations give us precisely the $2 \times s = n \times (n+1)$ that we need.

More formally, let us first construct the derivation that the loop body preserves the invariant:

$$\mathcal{H}_0 = \text{H-Conseq}\frac{\overset{(*)}{\models I \wedge i \neq n \Rightarrow I_2} \quad \text{H-Seq}\dfrac{\text{H-Assign}\dfrac{}{\{I_2\}\, i := i+1\, \{I_1\}} \quad \text{H-Assign}\dfrac{}{\{I_1\}\, s := s+i\, \{I\}}}{\{I_2\}\, i := i+1; s := s+i\, \{I\}} \quad \models I \Rightarrow I}{\{I \wedge i \neq n\}\, i := i+1; s := s+i\, \{I\}},$$

where

$$\begin{aligned} I &\equiv 2 \times s = i \times (i+1) \\ I_1 &\equiv I[(s+i)/s] \equiv 2 \times (s+i) = i \times (i+1) \\ I_2 &\equiv I_1[(i+1)/i] \equiv 2 \times (s+(i+1)) = (i+1) \times (i+1+1) \end{aligned}$$

It is immediate to check that the uses of H-Assign and H-Seq match their definitions. We also need to argue the semantic validity of the implication $(*)$, but that follows by straightforward equational reasoning: using just the first conjunct on the LHS, we calculate for the RHS:

$$2 \times (s+(i+1)) = 2 \times s + 2 \times (i+1) =^{(I)} i \times (i+1) + 2 \times (i+1) = (i+2) \times (i+1) = (i+1) \times (i+1+1)\,.$$

Then we can construct a derivation for the whole program:

$$\mathcal{H}_1 = \text{H-Seq}\frac{\text{H-Seq}\dfrac{\text{H-Assign}\dfrac{}{\{I_4\}\, s := 0\, \{I_3\}} \quad \text{H-Assign}\dfrac{}{\{I_3\}\, i := 0\, \{I\}}}{\{I_4\}\, s := 0; i := 0\, \{I\}} \quad \text{H-While}\dfrac{\overset{\mathcal{H}_0}{\{I \wedge i \neq n\}\, i := i+1; s := s+i\, \{I\}}}{\{I\}\, \textbf{while}\ i \neq n\ \textbf{do}\ (\cdots)\, \{I \wedge \neg(i \neq n)\}}}{\{I_4\}\, s := 0; i := 0; \textbf{while}\ i \neq n\ \textbf{do}\ (i := i+1; s := s+i)\, \{I \wedge \neg(i \neq n)\}},$$

where

$$\begin{aligned} I_3 &\equiv I[0/i] \equiv 2 \times s = 0 \times (0+1) \\ I_4 &\equiv I_3[0/s] \equiv 2 \times 0 = 0 \times (0+1) \end{aligned}$$

And finally, accounting for the original pre- and post-conditions:

$$\mathcal{H} = \text{H-Conseq}\frac{\overset{(**)}{\models \textbf{true} \Rightarrow I_4} \quad \overset{\mathcal{H}_1}{\{I_4\}\, SUM\, \{I \wedge \neg(i \neq n)\}} \quad \overset{(***)}{\models I \wedge \neg(i \neq n) \Rightarrow 2 \times s = n \times (n+1)}}{\{\textbf{true}\}\, SUM\, \{2 \times s = n \times (n+1)\}}.$$

Here, $(**)$ is immediate, because I_4 evidently holds in any store (both sides of the equation simplify to 0); and for $(***)$, we note that the second conjunct, $\neg(i \neq n)$ is logically equivalent to just $i = n$, and using that equality in the first conjunct, I, gives us precisely the desired conclusion.

Note how we have proved that *SUM* is partially correct (i.e., will never return incorrect results) even without assuming that n was non-negative, because when $n < 0$, the code just loop forever. Had we instead written the loop condition as **while** $i < n$ **do** $(\cdots)$, the program would no longer satisfy its original specification, because it would never enter

the loop for a negative n, and hence terminate immediately with $s = 0$, even if n were, say, -10. However, the modified program would still be correct with the precondition $n \geq 0$. To argue this, we would need a stronger invariant, which would now also include the conjunct $i \leq n$. This evidently holds before the loop (when $i = 0$), *if* we know that $n \geq 0$. Then, after the loop, we will have $i \leq n \wedge \neg(i < n)$, which gives us $i = n$ as before. And finally, we must prove that the invariant is preserved by the loop body, i.e., that after the assignment $i := i + 1$, we still have $i \leq n$. And for that, we'll now actually need that the loop test was true when we started executing the body, because $\models i < n \Rightarrow i + 1 \leq n$. Amending the formal derivations above accordingly is straightforward.

3.1.5 Proof presentation

Like for natural-deduction proofs, even though Hoare-logic derivations are conceptually tree-shaped objects, it quickly becomes impractical to render them graphically like that. However, unlike in first-order logic, there is exactly one dedicated rule for each language construct, so the shape of a Hoare proof follows the shape of the program almost exactly. The only exception is that, anywhere in the proof, we may use the consequence rule, without changing the command.

This property leads to an alternative style of writing down concrete Hoare proofs as "fully annotated programs", where *every* subcommand is written between two assertions (still in curly braces), specifying its pre- and post-condition in the tree-shaped derivation. (For the sequence rule, that means that the middle assertion C would be written twice, so we omit one of the copies, writing just $\{A\}\ c_0;\ \{C\}\ c_1\ \{B\}$ instead of $\{A\}\ c_0\ \{C\};\ \{C\}\ c_1\ \{B\}$) Uses of the consequence rule then correspond to writing two non-identical assertions next to each other. From such an annotated program, it is trivial to reconstruct the formal, tree-shaped derivation.

When assertion-annotating a program, we generally start at the bottom, with the desired final postcondition, and use the H-Skip, H-Assign, H-Seq, and H-If rules to syntactically "push" the postcondition backwards through each command. Occasionally, this process comes up against a fixed precondition, at which point we need to insert a semantic-reasoning (H-Conseq) step. In general, this only happens in three situations: (1) at the very top of the program, where we encounter the initial precondition; (2) immediately after a loop, where we meet the loop invariant and negated loop condition; and (3) at the top of a loop body, where the invariant as modified by the body comes up against the original invariant and (non-negated) loop condition. However, we are also free to explicitly invoke semantic reasoning elsewhere, whenever it may aid readability of the proof.

Regardless of the style of presentation, all non-trivial implications $\models A \Rightarrow A'$ used as premises in the consequence rule must be justified by annotations or footnotes to the proof, at the same level of detail and precision as in any mathematical proof. Often the two assertions are both iterated conjunctions, e.g., $A = (A_1 \wedge \cdots \wedge A_n)$ and $A' = (A'_1 \wedge \cdots \wedge A'_m)$; in this case, for each non-trivial conjunct in the conclusion A', we note from which conjuncts in the assumption A it follows, and how.

For example, consider the annotated multiplication program in Figure 3.1. Its overall specification says that if, intially, x contains i and y contains j, then if and when the program terminates, p will contain the product of i and j. (We cannot simply say

$\{x = i \wedge y = j\}$
if $0 \leq x$ **then**
  $\{x = i \wedge y = j \wedge 0 \leq x\}$
  $\{x \times y = i \times j \wedge 0 \leq x\}$ †1
  skip
  $\{x \times y = i \times j \wedge 0 \leq x\}$
else
  $\{x = i \wedge y = j \wedge \neg(0 \leq x)\}$
  $\{(0 - x) \times (0 - y) = i \times j \wedge 0 \leq 0 - x\}$ †2
  $(x := 0 - x;$
  $\{x \times (0 - y) = i \times j \wedge 0 \leq x\}$
  $y := 0 - y);$
  $\{x \times y = i \times j \wedge 0 \leq x\}$
$\{x \times y = i \times j \wedge 0 \leq x\}$
$\{x \times y + 0 = i \times j \wedge 0 \leq x\}$ †3
$p := 0;$
$\{x \times y + p = i \times j \wedge 0 \leq x\}$
while $1 \leq x$ **do**
  $\{x \times y + p = i \times j \wedge 0 \leq x \wedge 1 \leq x\}$
  $\{(x - 1) \times y + (p + y) = i \times j \wedge 0 \leq x - 1\}$ †4
  $(x := x - 1;$
  $\{x \times y + (p + y) = i \times j \wedge 0 \leq x\}$
  $p := p + y)$
  $\{x \times y + p = i \times j \wedge 0 \leq x\}$
$\{x \times y + p = i \times j \wedge 0 \leq x \wedge \neg(1 \leq x)\}$
$\{x \times y + p = i \times j \wedge x = 0\}$ †5
$\{p = i \times j\}$ †6

Figure 3.1: A fully annotated multiplication program

$p = x \times y$, because the original values of x and y may be gone by the time the program finishes.) Variables like i and j are sometimes called *ghost variables*: they may occur in assertions to specify the intended behavior of the progran, but are never read or modified by the executable code itself.

Overall, the program first ensures that x is non-negative, by negating both x and y if necessary; and then it computes $x \times y$ by adding y x times to p, which has been initialized to zero. The relationship between the variables before each loop iteration is expressed by the invariant (or "generalized snapshot", in Peter Naur's original terminology) in the annotation immediately above the **while**. The remaining internal annotations essentially follow from that one choice by applying the Hoare rules, as sketched above.

In this annotated program, there are 6 instances of semantic reasoning, each marked by a †:

1. $\overbrace{x = i}^{(1)} \wedge \overbrace{y = j}^{(2)} \wedge \overbrace{0 \leq x}^{(3)} \Rightarrow \overbrace{x \times y = i \times j}^{(a)} \wedge \overbrace{0 \leq x}^{(b)}.$

   Here, (a) follows directly from (1) and (2), while (b) is just (3).

2. $\overbrace{x = i}^{(1)} \wedge \overbrace{y = j}^{(2)} \wedge \overbrace{\neg(0 \leq x)}^{(3)} \Rightarrow \overbrace{(0 - x) \times (0 - y) = i \times j}^{(a)} \wedge \overbrace{0 \leq 0 - x}^{(b)}$.

   We get (a) from (1) and (2), because $(0-x)\times(0-y) = (-x)\times(-y) = x\times y = i\times j$; (b) follows from (3), because $\neg(0 \leq x) \Leftrightarrow 0 > x \Leftrightarrow 0 - x > 0 \Rightarrow 0 \leq 0 - x$.

3. $\overbrace{x \times y = i \times j}^{(1)} \wedge \overbrace{0 \leq x}^{(2)} \Rightarrow \overbrace{x \times y + 0 = i \times j}^{(a)} \wedge \overbrace{0 \leq x}^{(b)}$.

   (a) is just a trivial rephrasing of (1), while (b) is (2). (Such "cleanup" steps are generally better merged with some more essential instance of semantic reasoning, but may occasionally be inserted to prevent too much cruft from accumulating in the proof.)

4. $\overbrace{x \times y + p = i \times j}^{(1)} \wedge \overbrace{0 \leq x}^{(2)} \wedge \overbrace{1 \leq x}^{(3)} \Rightarrow \overbrace{(x - 1) \times y + (p + y) = i \times j}^{(a)} \wedge \overbrace{0 \leq x - 1}^{(b)}$.

   (a) follows from (1), since $(x-1)\times y+(p+y) = x\times y-y+p+y = x\times y+p = i\times j$. We get (b) from (3) by subtracting 1 from both sides of the inequality. (Note that the weaker assumption (2) is not used for anything.)

5. $\overbrace{x \times y + p = i \times j}^{(1)} \wedge \overbrace{0 \leq x}^{(2)} \wedge \overbrace{\neg(1 \leq x)}^{(3)} \Rightarrow \overbrace{x \times y + p = i \times j}^{(a)} \wedge \overbrace{x = 0}^{(b)}$

   (a) is just (1). (b) follows because (2) and (3) together give $0 \leq x < 1$, and since x is integer-valued, it must be equal to zero.

6. $\overbrace{x \times y + p = i \times j}^{(1)} \wedge \overbrace{x = 0}^{(2)} \Rightarrow \overbrace{p = i \times j}^{(a)}$. Here, we get (a) by simply substituting (2) in (1) and simplifying. (Normally, we'd want to combine the semantic-reasoning steps †5 and †6 into one. However, when each of them is so simple as to not require any further explanation, it may be acceptable to just present them as separate instances of consequence. Chains of more than two consequence steps should almost always be collapsed, though.)

3.1.6 Total correctness

Partial correctness of a program says that, *if* the program terminates, the result will satisfy the postcondition. A closely related notion, *total correctness*, additionally expresses that the program *will* terminate, assuming the initial state satisfied the precondition. Total-correctness triples are conventionally written with the assertions between square brackets, rather than curly braces:

$$\models [A]\, c\, [B] \iff \forall \sigma\colon \Sigma.\, (\sigma \models A) \Rightarrow \exists \sigma'\colon \Sigma.\, \langle c, \sigma\rangle \downarrow \sigma' \wedge (\sigma' \models B)\,.$$

Total correctness is usually established as a refinement of partial correctness, by also arguing that all **while**-loops in the program terminate. This can often be done fairly straightforwardly, by showing, for each loop, that some *upper bound* on the remaining number of iterations is strictly decreased by every execution of the loop body. For example, in the multiplication program of Figure 3.1, the value of x itself provides a suitable such bound for the loop in the second part.

In fact, the syntactic rules for $\vdash [A]\, c\, [B]$ are almost the same as those for $\vdash \{A\}\, c\, \{B\}$, except that the new H-WHILE rule includes a few extra requirements about the loop body. These additional conditions express precisely that the value of a clearly identified integer expression, often called the *variant* of the loop, gets strictly smaller after each iteration, while remaining non-negative at least until the loop condition becomes false. This property of the variant means that the loop body can at most be executed as many times as indicated by the value of the expression before the loop. For space reasons, we will not treat total correctness formally in these notes, however.

3.2 Soundness of the Hoare rules

We now aim to show that the proof system for Hoare logic is sound, i.e., that if a triple is provable, then it is also valid. For this, we will need a couple of lemmas:

Lemma 3.5 (Substitution for expressions) *If* $\langle a, \sigma\rangle \downarrow n$, *then* $\langle a_0[a/X], \sigma\rangle \downarrow n_0 \Leftrightarrow \langle a_0, \sigma[X \mapsto n]\rangle \downarrow n_0$.

Proof. We assume $\langle a, \sigma\rangle \downarrow n$, and proceed by a simple induction on the structure of a_0. The only interesting (but still immediate) cases are for $a_0 = X$ (to show: $\langle a, \sigma\rangle \downarrow n_0 \Leftrightarrow n_0 = n$); and $a_0 = X'$ for $X' \neq X$ (to show: $\langle X', \sigma\rangle \downarrow n_0 \Leftrightarrow n_0 = \sigma(X')$. ∎

Lemma 3.6 (Substitution for assertions) *If* $\langle a, \sigma\rangle \downarrow n$, *then* $(\sigma \models A[a/X]) \Leftrightarrow (\sigma[X \mapsto n] \models A)$.

Proof. Induction on the structure of A, using Lemma 3.5 for the case where A is $a_0 = a_1$ or $a_0 \leq a_1$. Note that the definition of substitution in assertions is capture-avoiding: in $(\forall Y.\, A)[a/X]$, when $Y \neq X$, we assume that Y also does not occur in a; otherwise, we must rename all occurrences of Y to a *fresh* (i.e., not occurring in A or a) location Y' before performing the substitution, to get $\forall Y'.\, A[Y'/Y][a/X]$. ∎

Theorem 3.7 (Soundness of Hoare logic) *If* $\vdash \{A\}\, c\, \{B\}$, *then* $\models \{A\}\, c\, \{B\}$.

Proof. Let $\mathcal{H}$ be the derivation of $\vdash \{A\}\, c\, \{B\}$. We shall show, by induction on $\mathcal{H}$, that for any $\sigma, \sigma' \in \Sigma$ such that $\sigma \models A$, and derivation $\mathcal{E}$ of $\langle c, \sigma\rangle \downarrow \sigma'$, we have $\sigma' \models B$:

- Case $\mathcal{H} = \text{H-SKIP}\dfrac{}{\{A\}\, \mathbf{skip}\, \{A\}}$, so $B = A$ and $c = \mathbf{skip}$. The only possible shape of $\mathcal{E}$ is then
$$\mathcal{E} = \text{EC-SKIP}\dfrac{}{\langle \mathbf{skip}, \sigma\rangle \downarrow \sigma}\,,$$
so $\sigma' = \sigma$. Thus, the assumption that $\sigma \models A$ also directly gives us that $\sigma' \models B$.
- Case $\mathcal{H} = \text{H-ASSIGN}\dfrac{}{\{B[a/X]\}\, X := a\, \{B\}}$, so $c = (X := a)$. Then the derivation $\mathcal{E}$ must be:
$$\mathcal{E} = \text{EC-ASSIGN}\dfrac{\begin{array}{c}\mathcal{E}_0\\ \langle a, \sigma\rangle \downarrow n\end{array}}{\langle X := a, \sigma\rangle \downarrow \sigma[X \mapsto n]}$$

for some n and $\mathcal{E}_0$, and so $\sigma' = \sigma[X \mapsto n]$. Since $A = B[a/X]$, Lemma 3.6($\Rightarrow$) on $\mathcal{E}_0$ and the assumption that $\sigma \models B[a/X]$, gives us precisely that $\sigma[X \mapsto n] \models B$.

- Case $\mathcal{H} = \text{H-SEQ}\dfrac{\overset{\mathcal{H}_0}{\{A\}\, c_0\, \{C\}} \quad \overset{\mathcal{H}_1}{\{C\}\, c_1\, \{B\}}}{\{A\}\, c_0; c_1\, \{B\}}$. Since $c = (c_0; c_1)$, we must have, for some σ'',

$$\mathcal{E} = \text{EC-SEQ}\frac{\overset{\mathcal{E}_0}{\langle c_0, \sigma\rangle \downarrow \sigma''} \quad \overset{\mathcal{E}_1}{\langle c_1, \sigma''\rangle \downarrow \sigma'}}{\langle c_0; c_1, \sigma\rangle \downarrow \sigma'}\,.$$

  Now, by IH on $\mathcal{H}_0$ with $\mathcal{E}_0$, we get that $\sigma'' \models C$. And thus, by IH on $\mathcal{H}_1$ with $\mathcal{E}_1$, we get $\sigma' \models B$, as required.

- Case $\mathcal{H} = \text{H-IF}\dfrac{\overset{\mathcal{H}_0}{\{A \wedge b\}\, c_0\, \{B\}} \qquad \overset{\mathcal{H}_1}{\{A \wedge \neg b\}\, c_1\, \{B\}}}{\{A\}\, \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1\, \{B\}}$.

  Since $c = \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$, $\mathcal{E}$ must use one of the two rules for executing **if**-commands. We thus have two subcases:

  - Subcase $\mathcal{E} = \text{EC-IFT}\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \textbf{true}} \qquad \overset{\mathcal{E}_1}{\langle c_0, \sigma\rangle \downarrow \sigma'}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \downarrow \sigma'}$. By Lemma 3.1($\Leftarrow$) on $\mathcal{E}_0$, we have $\sigma \models b$, and since we had already assumed $\sigma \models A$, we have $\sigma \models A \wedge b$. Thus, by IH on $\mathcal{H}_0$ with $\mathcal{E}_1$, we get $\sigma' \models B$, as required.
  - Subcase $\mathcal{E} = \text{EC-IFF}\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \textbf{false}} \qquad \overset{\mathcal{E}_1}{\langle c_1, \sigma\rangle \downarrow \sigma'}}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \downarrow \sigma'}$. The proof is analogous: we first see from $\mathcal{E}_0$ (by determinism of boolean expressions and Lemma 3.1($\Rightarrow$)) that $\sigma \not\models b$, so $\sigma \models A \wedge \neg b$, and thus by IH on $\mathcal{H}_1$ with $\mathcal{E}_1$, we again get $\sigma' \models B$.

- Case $\mathcal{H} = \text{H-WHILE}\dfrac{\overset{\mathcal{H}_0}{\{A \wedge b\}\, c_0\, \{A\}}}{\{A\}\, \textbf{while } b \textbf{ do } c_0\, \{A \wedge \neg b\}}$.

  Here we cannot simply do a split between two cases for $\mathcal{E}$. Instead, we first prove, by an *inner* induction on derivations, that for any σ'' such that $\sigma'' \models A$, and derivation $\mathcal{E}'$ of $\langle \textbf{while } b \textbf{ do } c_0, \sigma''\rangle \downarrow \sigma'$, we must have $\sigma' \models A \wedge \neg b$:

  - Case $\mathcal{E}' = \text{EC-WHILEF}\dfrac{\overset{\mathcal{E}'_0}{\langle b, \sigma''\rangle \downarrow \textbf{false}}}{\langle \textbf{while } b \textbf{ do } c_0, \sigma''\rangle \downarrow \sigma''}$. Here $\sigma' = \sigma''$, so $\sigma' \models A$. Also, by Lemma 3.1 on $\mathcal{E}'_0$, we get $\sigma' \not\models b$, i.e., that $\sigma' \models \neg b$. And thus, $\sigma' \models A \wedge \neg b$.
  - Case $\mathcal{E}' = \text{EC-WHILET}\dfrac{\overset{\mathcal{E}'_0}{\langle b, \sigma''\rangle \downarrow \textbf{true}} \qquad \overset{\mathcal{E}'_1}{\langle c_0, \sigma''\rangle \downarrow \sigma'''} \qquad \overset{\mathcal{E}'_2}{\langle \textbf{while } b \textbf{ do } c_0, \sigma'''\rangle \downarrow \sigma'}}{\langle \textbf{while } b \textbf{ do } c_0, \sigma''\rangle \downarrow \sigma'}$.
    First, Lemma 3.1 on $\mathcal{E}'_0$ gives us that $\sigma'' \models b$, and thus $\sigma'' \models A \wedge b$. Hence, by the *outer* IH on $\mathcal{H}_0$ with $\mathcal{E}'_1$, we get that $\sigma''' \models A$. But then, by the *inner* IH on $\mathcal{E}'_2$, we get that $\sigma' \models A \wedge \neg b$, as required.

To complete the case, we simply take σ'' as σ, and $\mathcal{E}'$ as $\mathcal{E}$ in the above result.

- Case $\mathcal{H} = \text{H-Conseq}\dfrac{\models A \Rightarrow A' \qquad \overset{\mathcal{H}'}{\{A'\}\, c\, \{B'\}} \qquad \models B' \Rightarrow B}{\{A\}\, c\, \{B\}}$.

  By definition of validity of assertions, we have in particular that $\sigma \models A \Rightarrow A'$, so since $\sigma \models A$, we also have $\sigma \models A'$. Thus, by IH on $\mathcal{H}'$ with $\mathcal{E}$, we get $\sigma' \models B'$. And since again $\sigma' \models B' \Rightarrow B$, we get the required $\sigma' \models B$. ∎

3.3 Completeness of the Hoare rules

We also have following converse of Theorem 3.7:

Theorem 3.8 (Completeness of Hoare logic) *If* $\models \{A\}\, c\, \{B\}$, *then* $\vdash \{A\}\, c\, \{B\}$.

This direction is rather more subtle, and was first proved by Stephen Cook in 1978. The proof (formulated in a denotational setting) can be found in *FSoPL* Section 7.2, which is not part of the syllabus for *Semantics and Types*. The result hinges crucially on the fact that the assertion language is *expressive*, i.e., that for any command c and postcondition B, there exists a *weakest liberal precondition* B^* (expressed entirely in the syntax of assertions, i.e., without reference to the semantics of IMP commands), such that

$$\sigma \models B^* \iff \forall \sigma'. \langle c, \sigma \rangle \downarrow \sigma' \Rightarrow (\sigma' \models B)\,.$$

That is, B^* says exactly what the starting state σ needs to satisfy in order to ensure that any final state σ' obtained by executing c in σ will satisfy B.

That such a syntactic B^* always exists, guarantees that we can find suitable invariants for all the loops in the program. It should be noted, though, that the B^* implicitly constructed in the proof of the theorem is often vastly more complicated than necessary, and hence of little practical utility. In particular, the required integer-arithmetic arguments for assertion validity in uses of the consequence rule become nearly unmanageable. It is usually much simpler to come up with suitable loop invariants by hand, and verify that they allow one to complete the proof.

Nevertheless, Theorem 3.8 is an extremely powerful result: it says that any program – no matter how trickily written – that manages to be partially correct with respect to a pre- and post-condition, can be formally verified to be so, using just the six Hoare rules. In other words, there is *in principle* no excuse for not properly proving correct any program we write.

3.3.1 Completeness and undecidability

Let be c be a command, and let the (necessarily finite) set $L = \{X_1, ..., X_k\}$ be all the locations occurring anywhere in c. It is easy to see that, as long as two starting states agree on just those k locations, executing c in one of them terminates iff it terminates in the other one, because c can neither read nor write any other part of the state.

Now, for any σ_0, let $is_L(\sigma_0)$ be the syntactic assertion "$X_1 = \overline{\sigma_0(X_1)} \wedge \cdots \wedge X_k = \overline{\sigma_0(X_k)}$". Then the statement $\models \{is_L(\sigma_0)\}\, c\, \{\mathbf{false}\}$ effectively says that c diverges from

state σ_0, because any σ, such that $\sigma \models is_L(\sigma_0)$, agrees with σ_0 on L; and any state σ' such that $\langle c, \sigma\rangle \downarrow \sigma'$ would satisfy $\sigma' \models \mathbf{false}$, which is impossible. Thus, for any c and σ_0, either (1) there exists a big-step derivation $\langle c, \sigma_0\rangle \downarrow \sigma'$ for some σ', or (2) we have $\models \{is_L(\sigma_0)\}\, c\, \{\mathbf{false}\}$, and hence (by completeness), there exists a Hoare-logic derivation of $\vdash \{is_L(\sigma_0)\}\, c\, \{\mathbf{false}\}$.

One might therefore think that a method for deciding *in principle* whether or not a program halts from a given starting state, would be to search in parallel among all well-formed derivations of the two judgments from cases (1) or (2); this search is *guaranteed* to eventually hit one (and, by soundness, only one) of the two possibilities.

The reason why this is not a contradiction to a well-known result from computability theory (undecidability of the Halting Problem) is that checking a Hoare-logic derivation is actually *not* an entirely mechanical process: in all uses of H-Conseq, the semantic premises $\models A \Rightarrow A'$ and $\models B' \Rightarrow B$ require reasoning about integer arithmetic. But as shown by Kurt Gödel, there can be no *sound and complete* proof system for just plain arithmetic assertions, i.e., a set of rules for $\vdash A$, such that $(\vdash A) \Leftrightarrow (\models A)$. So one sometimes says that Hoare logic is *relatively complete*: it can prove all valid partial-correctness triples, assuming we can argue the validity of all assertions appearing in the consequence rule.

3.3.2 Completeness and multiplication

Another curious fact about Theorem 3.8 is that its proof (specifically, the proof of expressivity of the assertion language) relies crucially on the fact that the grammar of arithmetic expressions includes multiplication, not only addition and subtraction. One can show that a slightly restricted language of assertions, with all the same logical connectives and quantifiers, only without a multiplication operator in the grammar of arithmetic expressions, actually has a sound and complete proof system, and there is an effective (if slow) algorithm for deciding whether any such assertion is valid or not. This was proved in 1929 by Mojżesz Presburger, and is by no means an obvious result.

Note that, even if we remove multiplication as a primitive arithmetic operator, we can still *compute* products, by using a simple while-loop (like the one in Figure 3.1). This means that any IMP program can be easily transformed into an equivalent program in restricted IMP without a built-in notion of multiplication. However, while Hoare logic would naturally remain sound for reasoning about programs in this restricted language, it would no longer be complete.

3.4 Exercises

3.1. Consider the following (correct, but ineptly written) division program DIV:

$$
\begin{array}{l}
r := n; q := 0; \\
\textbf{while } r > 0 \textbf{ do } (r := r - d; q := q + 1); \\
\textbf{if } r < 0 \textbf{ then } (r := r + d; q := q - 1) \textbf{ else skip}
\end{array}
$$

Derive the following triple in Hoare logic:

$$\{n \geq 0 \wedge d > 0\}\, DIV\, \{n = q \times d + r \wedge 0 \leq r \wedge r < d\}$$

Be sure to explain any non-trivial mathematical reasoning you use to justify semantic validity of assertions ($\models A \Rightarrow A'$) in the consequence rule.

Hint: Use the following assertion for the loop invariant: $n = q \times d + r \wedge r + d > 0$.

Hint 2: Do not try to construct a single big derivation tree for the entire program. Instead, introduce named subderivations proving correctness of fragments of the program, and show how they are put together into a complete derivation. Alternatively, you may use the fully-annotated style outlined in Section 3.1.5.

3.2. Suppose we extend the language with a **repeat**–**until** command like in Assignment 1. Its operational semantics is given by two rules:

$$\text{EC-RepeatT} : \frac{\langle c_0, \sigma\rangle \downarrow \sigma' \qquad \langle b, \sigma'\rangle \downarrow \mathbf{true}}{\langle \mathbf{repeat}\ c_0\ \mathbf{until}\ b, \sigma\rangle \downarrow \sigma'}$$

$$\text{EC-RepeatF} : \frac{\langle c_0, \sigma\rangle \downarrow \sigma'' \qquad \langle b, \sigma''\rangle \downarrow \mathbf{false} \qquad \langle \mathbf{repeat}\ c_0\ \mathbf{until}\ b, \sigma''\rangle \downarrow \sigma'}{\langle \mathbf{repeat}\ c_0\ \mathbf{until}\ b, \sigma\rangle \downarrow \sigma'}$$

and the axiomatic semantics is extended with an additional proof rule:

$$\text{H-Repeat} : \frac{\{A \vee (B \wedge \neg b)\}\, c_0\, \{B\}}{\{A\}\, \mathbf{repeat}\ c_0\ \mathbf{until}\ b\, \{B \wedge b\}}$$

Extend the proof of Theorem 3.7 to **repeat**-loops. *Hint:* Think carefully about how you formulate the inner induction, and make sure your choice actually works; it's easy to make a mistake here!

3.3. Show that, for any command c and assertion B, the following triple is provable in Hoare logic:

$$\vdash \{\mathbf{false}\}\, c\, \{B\}$$

Do the proof in two different ways:

(a) Using the Completeness theorem. (This gives a very short proof, but only works because the assertion language includes multiplication.)

(b) Using just the proof rules, without any reference to the operational semantics of IMP. (This takes a little more work, but gives a proof that does not depend on the set of available arithmetic operators.) Be very explicit about where you use the consequence rule in the derivations you construct. *Hint:* Show first, by structural induction on c, that $\vdash \{\mathbf{false}\}\, c\, \{\mathbf{false}\}$.