6 The axiomatic semantics of IMP

In this chapter we turn to the business of systematic verification of programs in **IMP**. The Hoare rules for showing the partial correctness of programs are introduced and shown sound. This involves extending the boolean expressions to a rich language of assertions about program states. The chapter concludes with an example of verification conducted within the framework of Hoare rules.

6.1 The idea

We turn to consider the problem of how to prove that a program we have written in **IMP** does what we require of it.

Let's start with a simple example of a program to compute the sum of the first hundred numbers, the naive way. Here is a program in **IMP** to compute $\sum_{1\leq m\leq 100} m$ (The notation $\sum_{1\leq m\leq 100} m$ means $1+2+\cdots+100$).

$$\begin{aligned}&S := 0;\\&N := 1;\\&(\textbf{while}\ \neg(N = 101)\ \textbf{do}\ S := S + N; N := N + 1)\end{aligned}$$

How would we prove that this program, when it terminates, is such that the value of S is $\sum_{1\leq m\leq 100} m$?

Of course one thing we could do would be to run it according to our operational semantics and see what we get. But suppose we change our program a bit, so that instead of "**while** $\neg(N = 101)$ **do** $\cdots$" we put "**while** $\neg(N = P+1)$ **do** $\cdots$" and imagine making some arbitrary assignment to P before we begin. In this case the resulting value of S after execution should be $\sum_{1\leq m\leq P} m$, no matter what the value of P. As P can take an infinite set of values we cannot justify this fact simply by running the program for all initial values of P. We need to be a little more clever, and abstract, and use some logic to reason about the program.

We'll end up with a formal proof system for proving properties of **IMP** programs, based on proof rules for each programming construct of **IMP**. Its rules are called Hoare rules or Floyd-Hoare rules. Historically R.W.Floyd invented rules for reasoning about flow charts, and later C.A.R.Hoare modified and extended these to give a treatment of a language like **IMP** but with procedures. Originally their approach was advocated not just for proving properties of programs but also as giving a method for explaining the meaning of program constructs; the meaning of a construct was specified in terms of "axioms" (more accurately rules) saying how to prove properties of it. For this reason, the approach is traditionally called axiomatic semantics.

For now let's not be too formal. Let's look at the program and reason informally about it, for the moment based on our intuitive understanding of how it behaves. Straightaway we see that the commands $S := 0; N := 1$ initialise the values in the locations. So we can annotate our program with a comment:

$$S := 0; N := 1$$
$$\{S = 0 \ \wedge \ N = 1\}$$
$$(\textbf{while } \neg(N = 101) \textbf{ do } S := S + N; N := N + 1)$$

with the understanding that $S = 0$ for example means the location S has value 0, as in the treatment of boolean expressions. We want a method to justify the final comment in:

$$S := 0; N := 1$$
$$\{S = 0 \ \wedge \ N = 1\}$$
$$(\textbf{while } \neg(N = 101) \textbf{ do } S := S + N; N := N + 1)$$
$$\{S = \sum_{1 \leq m \leq 100} m\}$$

—meaning that if $S = 0 \ \wedge \ N = 1$ before the execution of the while-loop then $S = \sum_{1 \leq m \leq 100} m$ after its execution.

Looking at the boolean, one fact we know holds after the execution of the while-loop is that we cannot have $N \neq 101$; because if we had $\neg(N = 101)$ then the while-loop would have continued running. So, at the end of its execution we know $N = 101$. But we want to know S!

Of course, with a simple program like this we can look and see what the values of S and N are the first time round the loop, $S = 1, N = 2$. And the second time round the loop $S = 1 + 2, N = 3 \cdots$ and so on, until we see the pattern: after the i th time round the loop $S = 1 + 2 + \cdots + i$ and $N = i + 1$. From which we see, when we exit the loop, that $S = 1 + 2 + \cdots + 100$, because when we exit $N = 101$.

At the beginning and end of each iteration of the while-loop we have

$$S = 1 + 2 + 3 + \cdots + (N - 1) \qquad (I)$$

which expresses the key relationship between the value at location S and the value at location N. The assertion I is called an *invariant* of the while-loop because it remains true under each iteration of the loop. So finally when the loop terminates I will hold at the end. We shall say more about invariants later.

For now it appears we can base a proof system on assertions of the form

$$\{A\}c\{B\}$$

where A and B are assertions like those we've already seen in **Bexp** and c is a command. The precise interpretation of such a compound assertion is this:

> for all states σ which satisfy A if the execution c from state σ terminates in state σ' then σ' satisfies B.

Put another way, $\{A\}c\{B\}$ means that any successful (*i.e.*, terminating) execution of c from a state satisfying A ends up in a state satisfying B. The assertion A is called the precondition and B the postcondition of the partial correctness assertion $\{A\}c\{B\}$.

Assertions of the form $\{A\}c\{B\}$ are called *partial correctness assertions* because they say nothing about the command c if it fails to terminate. As an extreme example consider

$$c \equiv \textbf{while true do skip}.$$

The execution of c from any state does not terminate. According to the interpretation we give above the following partial correctness assertion is valid:

$$\{\textbf{true}\}c\{\textbf{false}\}$$

simply because the execution of c does not terminate. More generally, because c loops, any partial correctness assertion $\{A\}c\{B\}$ is valid. Contrast this with another notion, that of total correctness. Sometimes people write

$$[A]c[B]$$

to mean that the execution of c from any state which satisfies A will terminate in a state which satisfies B. In this book we shall not be concerned much with total correctness assertions.

Warning: There are several different notations around for expressing partial and total correctness. When dipping into a book make doubly sure which notation is used there.

We have left several loose ends. For one, what kinds of assertions A and B do we allow in partial correctness assertions $\{A\}c\{B\}$? We say more in a moment, and turn to a more general issue.

The next issue can be regarded pragmatically as one of notation, though it can be viewed more conceptually as the semantics of assertions for partial correctness—see the "optional" Section 7.5 on denotational semantics using predicate transformers. Firstly let's introduce an abbreviation to mean the state σ satisfies assertion A, or equivalently the assertion A is true at state σ. We abbreviate this to:

$$\sigma \models A.$$

Of course, we'll need to define it, though we all have an intuitive idea of what it means. Consider our interpretation of a partial correctness assertion $\{A\}c\{B\}$. As a command c denotes a partial function from initial states to final states, the partial correctness assertion means:

$$\forall\sigma.\ (\sigma \models A\ \&\ \mathcal{C}[\![c]\!]\sigma \text{ is defined}) \Rightarrow \mathcal{C}[\![c]\!]\sigma \models B.$$

It is awkward working so often with the proviso that $\mathcal{C}[\![c]\!]\sigma$ is defined. Recall Chapter 5 on the denotational semantics of **IMP**. There we suggested that we use the symbol $\perp$ to represent an undefined state (or more strictly, null information about the state). For a command c we can write $\mathcal{C}[\![c]\!]\sigma = \perp$ whenever $\mathcal{C}[\![c]\!]\sigma$ is undefined, and, in accord with the composition of partial functions, take $\mathcal{C}[\![c]\!]\perp = \perp$. If we adopt the convention that $\perp$ satisfies any assertion, then our work on partial correctness becomes much simpler notationally. With the understanding that

$$\perp \models A$$

for any assertion A, we can describe the meaning of $\{A\}c\{B\}$ by

$$\forall\sigma \in \Sigma.\ \sigma \models A \Rightarrow \mathcal{C}[\![c]\!]\sigma \models B.$$

Because we are dealing with partial correctness this convention is consistent with our previous interpretation of partial correctness assertions. It's quite intuitive too; diverging computations denote $\perp$ and as we've seen they satisfy any postcondition.

6.2 The assertion language Assn

What kind of assertions do we wish to make about **IMP** programs? Because we want to reason about boolean expressions we'll certainly need to include all the assertions in **Bexp**. Because we want to make assertions using the quantifiers "$\forall i \cdots$" and "$\exists i \cdots$" we will need to work with extensions of **Bexp** and **Aexp** which include integer variables i over which we can quantify. Then, for example, we can say that an integer k is a multiple of another l by writing

$$\exists i.\ k = i \times l.$$

It will be shown in reasonable detail how to introduce integer variables and quantifiers for a particular language of assertions **Assn**. In principle, everything we'll do with assertions can be done in **Assn**—it is expressive enough—but in examples and exercises we will extend **Assn** in various ways, without being terribly strict about it. (For instance, in one example we'll use the notation $n! = n \times (n-1) \times \cdots \times 2 \times 1$ for the factorial function.)

Firstly, we extend **Aexp** to include integer variables i, j, k, *etc.*. This is done simply by extending the BNF description of **Aexp** by the additional rule which makes any integer variable $i, j, k, \cdots$ an integer expression. So the extended syntactic category **Aexpv** of arithmetic expressions is given by:

$$a ::= n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

where

n ranges over numbers, **N**

X ranges over locations, **Loc**

i ranges over integer variables, **Intvar**.

We extend boolean expressions to include these more general arithmetic expressions and quantifiers, as well as implication. The rules are:

$$A ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid A_0 \Rightarrow A_1 \mid \forall i.A \mid \exists i.A$$

We call the set of extended boolean assertions, **Assn**.

At school we have had experience in manipulating expressions like those above, though in those days we probably wrote mathematics down in a less abbreviated way, not using quantifiers for instance. When we encounter an integer variable i we think of it as standing for some arbitrary integer and do calculations with it like those "unknowns" $x, y, \cdots$ at school. An implication like $A_0 \Rightarrow A_1$ means if A_0 then A_1, and will be true if either A_0 is false or A_1 is true. We have used implication before in our mathematics, and now we have added it to our set of formal assertions **Assn**. We have a "commonsense" understanding of the expressions and assertions (and this should be all that is needed when doing the exercises). However, because we want to reason about proof systems based on assertions, not just examples, we shall be more formal, and give a theory of the meaning of expressions and assertions with integer variables. This is part of the predicate calculus.

6.2.1 Free and bound variables

We say an occurrence of an integer variable i in an assertion is *bound* if it occurs in the scope of an enclosing quantifier $\forall i$ or $\exists i$. If it is not bound we say it is *free*. For example, in

$$\exists i.\ k = i \times l$$

the occurrence of the integer variable i is bound, while those of k and l are free—the variables k and l are understood as standing for particular integers even if we are not precise about which. The same integer variable can have different occurrences in the same assertion one of which is free and another bound. For example, in

$$(i + 100 \leq 77) \wedge (\forall i.\ j + 1 = i + 3)$$

the first occurrence of i is free and the second bound, while the sole occurrence of j is free.

Although this informal explanation will probably suffice, we can give a formal definition using definition by structural induction. Define the set $\mathrm{FV}(a)$ of free variables of arithmetic expressions, extended by integer variables, $a \in \mathbf{Aexpv}$, by structural induction

$$\begin{aligned}
&\mathrm{FV}(n) = \mathrm{FV}(X) = \emptyset \\
&\mathrm{FV}(i) = \{i\} \\
&\mathrm{FV}(a_0 + a_1) = \mathrm{FV}(a_0 - a_1) = \mathrm{FV}(a_0 \times a_1) = \mathrm{FV}(a_0) \cup \mathrm{FV}(a_1)
\end{aligned}$$

for all $n \in \mathbf{N}, X \in \mathbf{Loc}, i \in \mathbf{Intvar}$, and $a_0, a_1 \in \mathbf{Aexpv}$. Define the free variables $\mathrm{FV}(A)$ of an assertion A by structural induction to be

$$\begin{aligned}
&\mathrm{FV}(\mathbf{true}) = \mathrm{FV}(\mathbf{false}) = \emptyset \\
&\mathrm{FV}(a_0 = a_1) = \mathrm{FV}(a_0 \leq a_1) = \mathrm{FV}(a_0) \cup \mathrm{FV}(a_1) \\
&\mathrm{FV}(A_0 \wedge A_1) = \mathrm{FV}(A_0 \vee A_1) = \mathrm{FV}(A_0 \Rightarrow A_1) = \mathrm{FV}(A_0) \cup \mathrm{FV}(A_1) \\
&\mathrm{FV}(\neg A) = \mathrm{FV}(A) \\
&\mathrm{FV}(\forall i.A) = \mathrm{FV}(\exists i.A) = \mathrm{FV}(A) \setminus \{i\}
\end{aligned}$$

for all $a_0, a_1 \in \mathbf{Aexpv}$, integer variables i and assertions A_0, A_1, A. Thus we have made precise the notion of free variable. Any variable which occurs in an assertion A and yet is not free is said to be bound. An assertion with no free variables is *closed.*

6.2.2 Substitution

We can picture an assertion A as

$$---i---i---$$

say, with free occurrences of the integer variable i. Let a be an arithmetic expression, which for simplicity we assume contains no integer variables. Then

$$A[a/i] \equiv ---a---a---$$

is the result of substituting a for i. If a contained integer variables then it might be necessary to rename some bound variables of A in order to avoid the variables in a becoming bound by quantifiers in A—this is how it's done for general substitutions.

We describe substitution more precisely in the simple case. Let i be an integer variable and a be an arithmetic expression without integer variables, and firstly define substitution into arithmetic expressions by the following structural induction:

$$\begin{aligned}
&n[a/i] \equiv n \qquad X[a/i] \equiv X\\
&j[a/i] \equiv j \qquad i[a/i] \equiv a\\
&(a_0 + a_1)[a/i] \equiv (a_0[a/i] + a_1[a/i])\\
&(a_0 - a_1)[a/i] \equiv (a_0[a/i] - a_1[a/i])\\
&(a_0 \times a_1)[a/i] \equiv (a_0[a/i] \times a_1[a/i])
\end{aligned}$$

where n is a number, X a location, j is an integer variable with $j \not\equiv i$ and $a_0, a_1 \in$ **Aexpv**. Now we define substitution of a for i in assertions by structural induction—remember a does not have any free variables so we need not take any precautions to avoid its variables becoming bound:

$$\begin{aligned}
&\mathbf{true}[a/i] \equiv \mathbf{true} \qquad \mathbf{false}[a/i] \equiv \mathbf{false}\\
&(a_0 = a_1)[a/i] \equiv (a_0[a/i] = a_1[a/i]) \qquad (a_0 \leq a_1)[a/i] \equiv (a_0[a/i] \leq a_1[a/i])\\
&(A_0 \wedge A_1)[a/i] \equiv (A_0[a/i] \wedge A_1[a/i]) \qquad (A_0 \vee A_1)[a/i] \equiv (A_0[a/i] \vee A_1[a/i])\\
&(\neg A)[a/i] \equiv \neg(A[a/i]) \qquad (A_0 \Rightarrow A_1)[a/i] \equiv (A_0[a/i] \Rightarrow A_1[a/i])\\
&(\forall j.A)[a/i] \equiv \forall j.(A[a/i]) \qquad (\forall i.A)[a/i] \equiv \forall i.A\\
&(\exists j.A)[a/i] \equiv \exists j.(A[a/i]) \qquad (\exists i.A)[a/i] \equiv \exists i.A
\end{aligned}$$

where $a_0, a_1 \in$ **Aexpv**, A_0, A_1 and A are assertions and j is an integer variable with $j \not\equiv i$.

As was mentioned, defining substitution $A[a/i]$ in the case where a contains free variables is awkward because it involves the renaming of bound variables. Fortunately we don't need this more complicated definition of substitution for the moment.

We use the same notation for substitution in place of a location X, so if an assertion $A \equiv ---X---$ then $A[a/X] = ---a---$, putting a in place of X. This time the (simpler) formal definition is left to the reader.

Exercise 6.1 Write down an assertion $A \in$ **Assn** with one free integer variable i which expresses that i is a prime number, *i.e.* it is required that:

$\sigma \models^I A$ iff $I(i)$ is a prime number.

□

Exercise 6.2 Define a formula $LCM \in$ **Assn** with free integer variables i, j and k, which means "i is the least common multiple of j and k," *i.e.* it is required that:

$$\sigma \models^I LCM \text{ iff } I(k) \text{ is the least common multiple of } I(i) \text{ and } I(j).$$

(Hint: The least common multiple of two numbers is the smallest non-negative integer divisible by both.) □

6.3 Semantics of assertions

Because arithmetic expressions have been extended to include integer variables, we cannot adequately describe the value of one of these new expressions using the semantic function $\mathcal{A}$ of earlier. We must first interpret integer variables as particular integers. This is the role of interpretations.

An *interpretation* is a function which assigns an integer to each integer variable *i.e.* a function $I : \mathbf{Intvar} \to \mathbf{N}$.

The meaning of expressions, Aexpv

Now we can define a semantic function $\mathcal{A}v$ which gives the value associated with an arithmetic expression with integer variables in a particular state in a particular interpretation; the value of an expression $a \in \mathbf{Aexpv}$ in a an interpretation I and a state σ is written as $\mathcal{A}v[\![a]\!]I\sigma$ or equivalently as $(\mathcal{A}v[\![a]\!](I))(\sigma)$. Define, by structural induction,

$$\begin{aligned}
&\mathcal{A}v[\![n]\!]I\sigma = n\\
&\mathcal{A}v[\![X]\!]I\sigma = \sigma(X)\\
&\mathcal{A}v[\![i]\!]I\sigma = I(i)\\
&\mathcal{A}v[\![a_0 + a_1]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma + \mathcal{A}v[\![a_1]\!]I\sigma\\
&\mathcal{A}v[\![a_0 - a_1]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma - \mathcal{A}v[\![a_1]\!]I\sigma\\
&\mathcal{A}v[\![a_0 \times a_1]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma \times \mathcal{A}v[\![a_1]\!]I\sigma
\end{aligned}$$

The definition of the semantics of arithmetic expressions with integer variables extends the denotational semantics given in Chapter 5 for arithmetic expressions without them.

Proposition 6.3 *For all $a \in$* **Aexp** *(without integer variables), for all states σ and for all interpretations I*

$$\mathcal{A}[\![a]\!]\sigma = \mathcal{A}v[\![a]\!]I\sigma.$$

Proof: The proof is a simple exercise in structural induction on arithmetic expressions. □

The meaning of assertions, Assn

Because we include integer variables, the semantic function requires an interpretation function as a further argument. The role of the interpretation function is solely to provide a value in **N** which is the interpretation of integer variables.

Notation: We use the notation $I[n/i]$ to mean the interpretation got from interpretation I by changing the value for integer-variable i to n *i.e.*

$$I[n/i](j) = \begin{cases} n & \text{if } j \equiv i, \\ I(j) & \text{otherwise.} \end{cases}$$

We could specify the meanings of assertions in **Assn** in the same way we did for expressions with integer variables, but this time taking the semantic function from assertions to functions which, given an interpretation and state as an argument, returned a truth value. We choose an alternative though equivalent course. Given an interpretation I we define directly those states which satisfy an assertion.

In fact, it is convenient to extend the set of states Σ to the set $\Sigma_\perp$ which includes the value $\perp$ associated with a nonterminating computation—so $\Sigma_\perp =_{def} \Sigma \cup \{\perp\}$. For $A \in$ **Assn** we define by structural induction when

$$\sigma \models^I A$$

for a state $\sigma \in \Sigma$, in an interpretation I, and then extend it so $\perp \models^I A$. The relation $\sigma \models^I A$ means state σ *satisfies* A in interpretation I, or equivalently, that assertion A is true at state σ, in interpretation I. By structural induction on assertions, for an interpretation I, we define for all $\sigma \in \Sigma$:

$$\begin{aligned}
&\sigma \models^I \mathbf{true}, \\
&\sigma \models^I (a_0 = a_1) \text{ if } \mathcal{A}v[\![a_0]\!]I\sigma = \mathcal{A}v[\![a_1]\!]I\sigma, \\
&\sigma \models^I (a_0 \le a_1) \text{ if } \mathcal{A}v[\![a_0]\!]I\sigma \le \mathcal{A}v[\![a_1]\!]I\sigma, \\
&\sigma \models^I A \wedge B \text{ if } \sigma \models^I A \text{ and } \sigma \models^I B, \\
&\sigma \models^I A \vee B \text{ if } \sigma \models^I A \text{ or } \sigma \models^I B, \\
&\sigma \models^I \neg A \text{ if not } \sigma \models^I A, \\
&\sigma \models^I A \Rightarrow B \text{ if (not } \sigma \models^I A) \text{ or } \sigma \models^I B, \\
&\sigma \models^I \forall i.A \text{ if } \sigma \models^{I[n/i]} A \text{ for all } n \in \mathbf{N}, \\
&\sigma \models^I \exists i.A \text{ if } \sigma \models^{I[n/i]} A \text{ for some } n \in \mathbf{N} \\
&\perp \models^I A.
\end{aligned}$$

Note that, not $\sigma \models^I A$ is generally written as $\sigma \not\models^I A$.

The above tells us formally what it means for an assertion to be true at a state once we decide to interpret integer variables in a particular way fixed by an interpretation. The semantics of boolean expressions provides another way of saying what it means for certain kinds of assertions to be true or false at a state. We had better check that the two ways agree.

Proposition 6.4 *For $b \in$ **Bexp**, $\sigma \in \Sigma$,*

$$\mathcal{B}[\![b]\!]\sigma = \mathbf{true} \textit{ iff } \sigma \models^I b, \textit{ and}$$
$$\mathcal{B}[\![b]\!]\sigma = \mathbf{false} \textit{ iff } \sigma \not\models^I b$$

for any interpretation I.

Proof: The proof is by structural induction on boolean expressions, making use of Proposition 6.3. □

Exercise 6.5 Prove the above proposition. □

Exercise 6.6 Prove by structural induction on expressions $a \in$ **Aexpv** that

$$\mathcal{A}v[\![a]\!]I[n/i]\sigma = \mathcal{A}v[\![a[n/i]]\!]I\sigma.$$

(Note that n occurs as an element of **N** on the left and as the corresponding number in **N** on the right.)

By using the fact above, prove

$$\sigma \models^I \forall i.A \quad \text{iff} \quad \sigma \models^I A[n/i] \text{ for all } n \in \mathbf{N} \quad \text{and}$$
$$\sigma \models^I \exists i.A \quad \text{iff} \quad \sigma \models^I A[n/i] \text{ for some } n \in \mathbf{N}.$$

□

The extension of an assertion

Let I be an interpretation. Often when establishing properties about assertions and partial correctness assertions it is useful to consider the extension of an assertion with respect to I *i.e.* the set of states at which the assertion is true.

Define the extension of A, an assertion, with respect to an interpretation I to be

$$A^I = \{\sigma \in \Sigma_\perp \mid \sigma \models^I A\}.$$

Partial correctness assertions

A partial correctness assertion has the form

$$\{A\}c\{B\}$$

where $A, B \in$ **Assn** and $c \in$ **Com**. Note that partial correctness assertions are not in **Assn**.

Let I be an interpretation. Let $\sigma \in \Sigma_\perp$. We define the satisfaction relation between states and partial correctness assertions, with respect to I, by

$$\sigma \models^I \{A\}c\{B\} \text{ iff } (\sigma \models^I A \Rightarrow \mathcal{C}[\![c]\!]\sigma \models^I B).$$

for an interpretation I. In other words, a state σ satisfies a partial correctness assertion $\{A\}c\{B\}$, with respect to an interpretation I, iff any successful computation of c from σ ends up in a state satisfying B.

Validity

Let I be an interpretation. Consider $\{A\}c\{B\}$. We are not so much interested in this partial correctness assertion being true at a particular state so much as whether or not it is true at all states *i.e.*

$$\forall \sigma \in \Sigma_\perp.\ \sigma \models^I \{A\}c\{B\},$$

which we can write as

$$\models^I \{A\}c\{B\},$$

expressing that the partial correctness assertion is valid with respect to the interpretation I, because $\{A\}c\{B\}$ is true regardless of which state we consider. Further, consider *e.g.*

$$\{i < X\}X := X + 1\{i < X\}$$

We are not so much interested in the particular value associated with i by the interpretation I. Rather we are interested in whether or not it is true at all states for all interpretations I. This motivates the notion of *validity.* Define

$$\models \{A\}c\{B\}$$

to mean for all interpretations I and all states σ

$$\sigma \models^I \{A\}c\{B\}.$$

When $\models \{A\}c\{B\}$ we say the partial correctness assertion $\{A\}c\{B\}$ is *valid.*

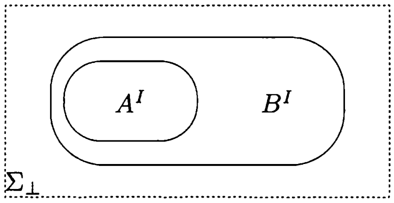
Similarly for any assertion A, write $\models A$ iff for all interpretations I and states σ, $\sigma \models^I A$. Then say A is *valid.*

Warning: Although closely related, our notion of validity is not the same as the notion of validity generally met in a standard course on predicate calculus or "logic programming." There an assertion is called valid iff for all interpretations for operators like $+, \times \cdots$, numerals $0, 1, \cdots$, as well as free variables, the assertion turns out to be true. We are not interested in arbitrary interpretations in this general sense because **IMP** programs operate on states based on locations with the standard notions of integer and integer operations. To distinguish the notion of validity here from the more general notion we could call our notion arithmetic-validity, but we'll omit the "arithmetic."

Example: Suppose $\models (A \Rightarrow B)$. Then for any interpretation I,

$$\forall \sigma \in \Sigma.\ ((\sigma \models^I A) \Rightarrow (\sigma \models^I B))$$

i.e. $A^I \subseteq B^I$. In a picture:



So $\models (A \Rightarrow B)$ iff for all interpretations I, all states which satisfy A also satisfy B. □

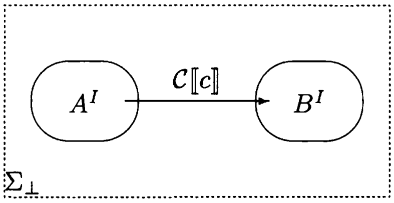
Example: Suppose $\models \{A\}c\{B\}$. Then for any interpretation I,

$$\forall \sigma \in \Sigma.\ ((\sigma \models^I A) \Rightarrow (\mathcal{C}[\![c]\!]\sigma \models^I B)),$$

i.e. the image of A under $\mathcal{C}[\![c]\!]$ is included in B *i.e.*

$$\mathcal{C}[\![c]\!]A^I \subseteq B^I.$$

In a picture:



So $\models \{A\}c\{B\}$ iff for all interpretations I, if c is executed from a state which satisfies A then if its execution terminates in a state that state will satisfy B.[1] □

Exercise 6.7 In an earlier exercise it was asked to write down an assertion $A \in \mathbf{Assn}$ with one free integer variable i expressing that i was prime. By working through the appropriate cases in the definition of the satisfaction relation $\models^I$ between states and assertions, trace out the argument that $\models^I A$ iff $I(i)$ is indeed a prime number. □

6.4 Proof rules for partial correctness

We present proof rules which generate the valid partial correctness assertions. The proof rules are syntax-directed; the rules reduce proving a partial correctness assertion of a compound command to proving partial correctness assertions of its immediate subcommands. The proof rules are often called *Hoare rules* and the proof system, consisting of the collection of rules, *Hoare logic.*

Rule for **skip**:

$$\{A\}\mathbf{skip}\{A\}$$

Rule for assignments:

$$\{B[a/X]\}X := a\{B\}$$

Rule for sequencing:

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$$

Rule for conditionals:

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\{B\}}$$

Rule for while loops:

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

Rule of consequence:

$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}}$$

[1]The picture suggests, incorrectly, that the extensions of assertions A^I and B^I are disjoint; they will both always contain $\perp$, and perhaps have other states in common.

Being rules, there is a notion of derivation for the Hoare rules. In this context the Hoare rules are thought of as a proof system, derivations are called *proofs* and any conclusion of a derivation a *theorem*. We shall write $\vdash \{A\}c\{B\}$ when $\{A\}c\{B\}$ is a theorem.

The rules are fairly easy to understand, with the possible exception of the rules for assignments and while-loops. If an assertion is true of the state before the execution of **skip** it is certainly true afterwards as the state is unchanged. This is the content of the rule for **skip**.

For the moment, to convince that the rule for assignments really is the right way round, it can be tried for a particular assertion such as $X = 3$ for the simple assignment like $X := X + 3$.

The rule for sequential compositions expresses that if $\{A\}c_0\{C\}$ and $\{C\}c_1\{B\}$ are valid then so is $\{A\}c_0; c_1\{B\}$: if a successful execution of c_0 from a state satisfying A ends up in one satisfying C and a successful execution of c_1 from a state satisfying C ends up in one satisfying B, then any successful execution of c_0 followed by c_1 from a state satisfying A ends up in one satisfying B.

The two premises in the rule for conditionals cope with two arms of the conditional.

In the rule for while-loops **while** b **do** c, the assertion A is called the *invariant* because the premise, that $\{A \wedge b\}c\{A\}$ is valid, says that the assertion A is preserved by a full execution of the body of the loop, and in a while loop such executions only take place from states satisfying b. From a state satisfying A either the execution of the while-loop diverges or a finite number of executions of the body are performed, each beginning in a state satisfying b. In the latter case, as A is an invariant the final state satisfies A and also $\neg b$ on exiting the loop.

The consequence rule is peculiar because the premises include valid implications. Any instance of the consequence rule has premises including ones of the form $\models (A \Rightarrow A')$ and $\models (B' \Rightarrow B)$ and so producing an instance of the consequence rule with an eye to applying it in a proof depends on first showing assertions $(A \Rightarrow A')$ and $(B' \Rightarrow B)$ are valid. In general this can be a very hard task—such implications can express complicated facts about arithmetic. Fortunately, because programs often do not involve deep mathematical facts, the demonstration of these validities can frequently be done with elementary mathematics.

6.5 Soundness

We consider for the Hoare rules two very general properties of logical systems:

Soundness: Every rule should preserve validity, in the sense that if the assumptions in the rule's premise is valid then so is its conclusion. When this holds of a rule it is called *sound.* When every rule of a proof system is sound, the proof system itself is said to be *sound.* It follows then by rule-induction that every theorem obtained from the proof system of Hoare rules is a valid partial correctness assertion. (The comments which follow the rules are informal arguments for the soundness of some of the rules.)

Completeness: Naturally we would like the proof system to be strong enough so that all valid partial correctness assertions can be obtained as theorems. We would like the proof system to be *complete* in this sense. (There are some subtle issues here which we discuss in the next chapter.)

The proof of soundness of the rules depends on some facts about substitution.

Lemma 6.8 *Let I be an interpretation. Let $a, a_0 \in$ **Aexpv**. Let $X \in$ **Loc**. Then for all interpretations I and states σ*

$$\mathcal{A}v[\![a_0[a/X]]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma[\mathcal{A}v[\![a]\!]I\sigma/X].$$

Proof: The proof is by structural induction on a_0—exercise! □

Lemma 6.9 *Let I be an interpretation. Let $B \in$ **Assn**, $X \in$ **Loc** and $a \in$ **Aexp**. For all states $\sigma \in \Sigma$*

$$\sigma \models^I B[a/X] \quad \textit{iff} \quad \sigma[\mathcal{A}[\![a]\!]\sigma/X] \models^I B.$$

Proof: The proof is by structural induction on B—exercise! □

Exercise 6.10 Provide the proofs for the lemmas above. □

Theorem 6.11 *Let $\{A\}c\{B\}$ be a partial correctness assertion.*
If $\vdash \{A\}c\{B\}$ then $\models \{A\}c\{B\}$.

Proof: Clearly if we can show each rule is sound (*i.e.* preserves validity in the sense that if its premise consists of valid assertions and partial correctness assertions then so is its conclusion) then by rule-induction we can see that every theorem is valid.

The rule for **skip**: Clearly $\models \{A\}$**skip**$\{A\}$ so the rule for **skip** is sound.

The rule for assignments: Assume $c \equiv (X := a)$. Let I be an interpretation. We have $\sigma \models^I B[a/X]$ iff $\sigma[\mathcal{A}[\![a]\!]\sigma/X] \models^I B$, by Lemma 6.9. Thus

$$\sigma \models^I B[a/X] \Rightarrow \mathcal{C}[\![X := a]\!]\sigma \models^I B,$$

and hence $\models \{B[a/X]\}X := a\{B\}$, showing the soundness of the assignment rule.

The rule for sequencing: Assume $\models \{A\}c\{_{\}}0C$ and $\models \{C\}c\{_{\}}1B$. Let I be an interpretation. Suppose $\sigma \models^I A$. Then $\mathcal{C}[\![c_0]\!]\sigma \models^I C$ because $\models^I \{A\}c\{_{\}}0C$. Also $\mathcal{C}[\![c_1]\!](\mathcal{C}[\![c_0]\!]\sigma) \models_I B$ because $\models^I \{C\}c\{_{\}}1B$. Hence $\models \{A\}c_0; c_1\{B\}$.

The rule for conditionals: Assume $\models \{A \wedge b\}c_0\{B\}$ and $\models \{A \wedge \neg b\}c_1\{B\}$. Let I be an interpretation. Suppose $\sigma \models_I A$. Either $\sigma \models_I b$ or $\sigma \models_I \neg b$. In the former case $\sigma \models_I A \wedge b$ so $\mathcal{C}[\![c_0]\!]\sigma \models^I B$, as $\models^I \{A \wedge b\}c_0\{B\}$. In the latter case $\sigma \models_I A \wedge \neg b$ so $\mathcal{C}[\![c_1]\!]\sigma \models^I B$, as $\models^I \{A \wedge \neg b\}c_1\{B\}$. This ensures $\models \{A\}$**if** b **then** c_0 **else** $c_1\{B\}$.

The rule for while-loops: Assume $\models \{A \wedge b\}c\{A\}$, *i.e.* A is an invariant of

$$w \equiv \textbf{while } b \textbf{ do } c.$$

Let I be an interpretation. Recall that $\mathcal{C}[\![w]\!] = \bigcup_{n\in\omega} \theta_n$ where

$$\theta_0 = \emptyset,$$
$$\theta_{n+1} = \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{true} \ \&\ (\sigma,\sigma') \in \theta_n \circ \mathcal{C}[\![c]\!]\} \cup \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{false}.\}$$

We shall show by mathematical induction that $P(n)$ holds where

$$P(n) \iff {}_{def}\forall \sigma,\sigma' \in \Sigma.\ (\sigma,\sigma') \in \theta_n \ \& \\ \sigma \models^I A \quad \Rightarrow \quad \sigma' \models^I A \wedge \neg b$$

for all $n \in \omega$. It then follows that

$$\sigma \models^I A \Rightarrow \mathcal{C}[\![w]\!]\sigma \models^I A \wedge \neg b$$

for all states σ, and hence that $\models \{A\}w\{A \wedge \neg b\}$, as required.
Base case $n = 0$: When $n = 0$, $\theta_0 = \emptyset$ so that induction hypothesis $P(0)$ is vacuously true.
Induction Step: We assume the induction hypothesis $P(n)$ holds for $n \geq 0$ and attempt to prove $P(n+1)$. Suppose $(\sigma,\sigma') \in \theta_{n+1}$ and $\sigma \models^I A$. Either
(i) $\mathcal{B}[\![b]\!]\sigma = \textbf{true}$ and $(\sigma,\sigma') \in \theta_n \circ \mathcal{C}[\![c]\!]$, or
(ii) $\mathcal{B}[\![b]\!]\sigma = \textbf{false}$ and $\sigma' = \sigma$.

We show in either case that $\sigma' \models^I A \wedge \neg b$.
Assume (i). As $\mathcal{B}[\![b]\!]\sigma = \mathbf{true}$ we have $\sigma \models^I b$ and hence $\sigma \models^I A \wedge b$. Also $(\sigma, \sigma'') \in \mathcal{C}[\![c]\!]$ and $(\sigma'', \sigma') \in \theta_n$ for some state σ''. We obtain $\sigma'' \models^I A$, as $\models \{A \wedge b\}c\{A\}$. From the assumption $P(n)$, we obtain $\sigma' \models^I A \wedge \neg b$.
Assume (ii). As $\mathcal{B}[\![b]\!]\sigma = \mathbf{false}$ we have $\sigma \models^I \neg b$ and hence $\sigma \models^I A \wedge \neg b$. But $\sigma' = \sigma$.

This establishes the induction hypothesis $P(n+1)$. By mathematical induction we conclude $P(n)$ holds for all n. Hence the rule for while loops is sound.

The consequence rule: Assume $\models (A \Rightarrow A')$ and $\models \{A'\}c\{B'\}$ and $\models (B' \Rightarrow B)$. Let I be an interpretation. Suppose $\sigma \models^I A$. Then $\sigma \models^I A'$, hence $\mathcal{C}[\![c]\!]\sigma \models^I B'$ and hence $\mathcal{C}[\![c]\!]\sigma \models^I B$. Thus $\models \{A\}c\{B\}$. The consequence rule is sound.

By rule-induction, every theorem is valid. □

Exercise 6.12 Prove the above using only the operational semantics, instead of the denotational semantics. What proof method is used for the case of while-loops? □

6.6 Using the Hoare rules—an example

The Hoare rules determine a notion of formal proof of partial correctness assertions through the idea of derivation. This is useful in the mechanisation of proofs. But in practice, as human beings faced with the task of verifying a program, we need not be so strict and can argue at a more informal level when using the Hoare rules. (Indeed working with the more formal notion of derivation might well distract from getting the proof; the task of producing the formal derivation should be delegated to a proof assistant like LCF or HOL [74], [43].)

As an example we show in detail how to use the Hoare rules to verify that the command

$$w \equiv (\mathbf{while}\ X > 0\ \mathbf{do}\ Y := X \times Y; X := X - 1)$$

does indeed compute the factorial function $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$, with $0!$ understood to be 1, given that $X = n$, a nonnegative number, and $Y = 1$ initially.[2]

More precisely, we wish to prove:

$$\{X = n \wedge n \geq 0 \wedge Y = 1\}w\{Y = n!\}.$$

To prove this we must clearly invoke the proof rule for while-loops which requires an invariant. Take

$$I \equiv (Y \times X! = n! \wedge X \geq 0).$$

[2]For this example, we imagine our syntax of programs and assertions to be extended to include $>$ and the factorial function which strictly speaking do not appear in the boolean and arithmetic expressions defined earlier.

We show I is indeed an invariant *i.e.*

$$\{I \wedge X > 0\}Y := X \times Y; X := X - 1\{I\}.$$

From the rule for assignment we have

$$\{I[(X-1)/X]\}X := X - 1\{I\}$$

where $I[(X-1)/X] \equiv (Y \times (X-1)! = n! \wedge (X-1) \geq 0)$. Again by the assignment rule:

$$\{X \times Y \times (X-1)! = n! \wedge (X-1) \geq 0\}Y := X \times Y\{I[(X-1)/X]\}.$$

Thus, by the rule for sequencing,

$$\{X \times Y \times (X-1)! = n! \wedge (X-1) \geq 0\}Y := X \times Y; X := (X-1)\{I\}.$$

Clearly

$$\begin{aligned} I \wedge X > 0 &\Rightarrow Y \times X! = n! \wedge X \geq 0 \wedge X > 0 \\ &\Rightarrow Y \times X! = n! \wedge X \geq 1 \\ &\Rightarrow X \times Y \times (X-1)! = n! \wedge (X-1) \geq 0. \end{aligned}$$

Thus by the consequence rule

$$\{I \wedge X > 0\}Y := X \times Y; X := (X-1)\{I\}$$

establishing that I is an invariant.

Now applying the rule for while-loops we obtain

$$\{I\}w\{I \ \wedge \ X \not> 0\}.$$

Clearly $(X = n) \wedge (n \geq 0) \wedge (Y = 1) \Rightarrow I$, and

$$\begin{aligned} I \ \wedge \ X \not> 0 &\Rightarrow Y \times X! = n! \wedge X \geq 0 \wedge X \not> 0 \qquad (*) \\ &\Rightarrow Y \times X! = n! \wedge X = 0 \\ &\Rightarrow Y \times 0! = Y = n! \end{aligned}$$

Thus by the consequence rule we conclude

$$\{(X = n) \wedge (Y = 1)\}w\{Y = n!\}.$$

There are a couple of points to note about the proof given in the example. Firstly, in dealing with a chain of commands composed in sequence it is generally easier to proceed in a right-to-left manner because the rule for assignment is of this nature. Secondly, our choice of I may seem unduly strong. Why did we include the assertion $X \geq 0$ in the invariant? Notice where it was used, at $(*)$, and without it we could not have deduced that on exiting the while-loop the value of X is 0. In getting invariants to prove what we want they often must be strengthened. They are like induction hypotheses. One obvious way to strengthen an invariant is to specify the range of the variables and values at the locations as tightly as possible. Undoubtedly, a common difficulty in examples is to get stuck on proving the "exit conditions". In this case, it is a good idea to see how to strengthen the invariant with information about the variables and locations in the boolean expression.

Thus it is fairly involved to show even trivial programs are correct. The same is true, of course, for trivial bits of mathematics, too, if one spells out all the details in a formal proof system. One point of formal proof systems is that proofs of properties of programs can be automated as in *e.g.*[74][41]—see also Section 7.4 on verification conditions in the next chapter. There is another method of application of such formal proof systems which has been advocated by Dijkstra and Gries among others, and that is to use the ideas in the study of program correctness in the design and development of programs. In his book "The Science of Programming" [44], Gries says

> "the study of program correctness proofs has led to the discovery and elucidation of methods for developing programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way!"

See Gries' book for many interesting examples of this approach.

Exercise 6.13 Prove, using the Hoare rules, the correctness of the partial correctness assertion:

$$\begin{array}{l}\{1 \leq N\}\\ P := 0;\\ C := 1;\\ (\textbf{while } C \leq N \textbf{ do } P := P + M; C := C + 1)\\ \{P = M \times N\}\end{array}$$

□

Exercise 6.14 Find an appropriate invariant to use in the while-rule for proving the following partial correctness assertion:

$$\{i = Y\}\textbf{while } \neg(Y = 0) \textbf{ do } Y := Y - 1; X := 2 \times X\{X = 2^i\}$$

□

Exercise 6.15 Using the Hoare rules, prove that for integers n, m,

$$\{X = m \wedge Y = n \wedge Z = 1\}c\{Z = m^n\}$$

where c is the while-program

```
while $\neg(Y = 0)$ do
((while $even(Y)$ do $X := X \times X; Y := Y/2$);
$Z := Z \times X; Y := Y - 1$)
```

with the understanding that $Y/2$ is the integer resulting from dividing the contents of Y by 2, and $even(Y)$ means the content of Y is an even number.
(Hint: Use $m^n = Z \times X^Y$ as the invariants.) □

Exercise 6.16
(i) Show that the greatest common divisor, $\gcd(n, m)$ of two positive numbers n, m satisfies:

$$(a)\ n > m \Rightarrow \gcd(n, m) = \gcd(n - m, m)$$
$$(b)\ \gcd(n, m) = \gcd(m, n)$$
$$(c)\ \gcd(n, n) = n.$$

(ii) Using the Hoare rules prove

$$\{N = n \wedge M = m \wedge 1 \leq n \wedge 1 \leq m\}\text{Euclid}\{X = \gcd(n, m)\}$$

where

```
Euclid ≡ while $\neg(M = N)$ do
             if $M \leq N$
              then $N := N - M$
              else $M := M - N$.
```

□

Exercise 6.17 Provide a Hoare rule for the repeat construct and prove it sound.
(*cf.* Exercise 5.9.) □

6.7 Further reading

The book [44] by Gries has already been mentioned. Dijkstra's "A discipline of programming" [36] has been very influential. A more elementary book in the same vein is Backhouse's "Program construction and verification" [12]. A recent book which is recommended is Cohen's "Programming in the 1990's" [32]. A good book with many exercises is Alagić and Arbib's "The design of well-structured and correct programs" [5]. An elementary treatment of Hoare logic with a lot of informative discussion can be found in Gordon's recent book [42]. Alternatives to this book's treatment, concentrating more on semantic issues than the other references, can be found in de Bakker's "Mathematical theory of program correctness" [13] and Loeckx and Sieber's "The foundations of program verification" [58].