So far we have covered *operational semantics*, which models computation happening on an abstract machine and talks about how state changes from one step of execution to the next. We have also talked about *denotational semantics*, which models computation as mathematical functions from inputs to outputs. In both of these there is a well-defined notion of the state of the system, and we take great pains to say exactly what that state is and how the program changes it.

In *axiomatic semantics*, on the other hand, we do not particularly care about what the states actually are, we only care about properties that we can observe about the states and how the programs impact those properties. This approach emphasizes the relationship between the properties of the input (preconditions) and properties of the output (postconditions). It is useful for specifying what a program is supposed to do and talk about a program's correctness with respect to that specification.

1 Preconditions and Postconditions

The *preconditions* and *postconditions* of a program say what is true before and after the program executes, respectively. Often the correctness of the program is specified in these terms. Typically this is expressed as a contract: as long as the caller guarantees that the initial state satisfies some set of preconditions, then the program will guarantee that the final state will satisfy some desired set of postconditions. Axiomatic semantics attempts to say exactly what preconditions are necessary for ensuring a given set of postconditions.

1.1 An Example

Consider the following Imp program designed to compute x^p:

$$
\begin{array}{l}
y := 1 \ ; \\
i := 0 \ ; \\
\mathsf{while}\ (i < p)\ \mathsf{do}\ \{ \\
\quad y := y * x \ ; \\
\quad i := i + 1 \\
\}
\end{array}
$$

The desired postcondition would be $y = x^p$. That is, the value of y in the final state of the program is precisely the initial value of x raised to the power of the initial value of p. One essential precondition we need is $p \geq 0$, because otherwise the program will halt immediately with $y = 1 \neq x^p$. Note that $p > 0$ would also be sufficient to guarantee the program halts with the correct output, but this is a stronger condition (is satisfied by fewer states, has more logical consequences).

$$
\underbrace{p > 0}_{\text{stronger}} \implies \underbrace{p \geq 0}_{\text{weaker}}
$$

The weaker precondition is better. It is less restrictive about the input states—in this case starting values of p—on which it ensures correctness. Typically, given a postcondition expressing a desired property of the output state, we would like to know the *weakest precondition* that guarantees the program halts and satisfies that postcondition upon termination.

1.2 Partial vs Total Correctness

There are two different notions of correctness, partial and total, that differ based on whether they allow nontermination.

- *Partial correctness* requires that a program behave correctly whenever it terminates. This is what we will be focusing on in this discussion.
- *Total correctness* requires that a program behave correctly *and terminate*. Total correctness is a much stronger requirement.

2 Hoare Logic

Hoare logic is a common way of specifying and reasoning about relationships between pre- and postconditions. It is named for its inventor Sir Charles Antony Richard "Tony" Hoare (b. 1934), an early pioneer of computer science who, among other things, also invented quicksort and has referred to "inventing" the null reference as "my billion-dollar mistake."

Hoare logic consists of *Hoare triples* of a precondition φ, a program c, and a postcondition ψ that form a partial correctness assertion (PCA) and are usually written

$$\{\varphi\}\ c\ \{\psi\}\,.$$

Informally, this triple means "if φ holds before execution of c and c terminates, then ψ will hold upon termination."

To define these assertions, we need some sort of logical language in which to write the pre- and postconditions. A simple option could be simple boolean expressions in the language itself (so **BExp** in Imp), or a general first-order logic. Given such an underlying logic and a language of programs, we can define a semantic meaning of a Hoare triple in terms of some existing semantics for programs, and we can define a set of proof rules for Hoare logic.

2.1 Hoare Logic for Imp

To build a logic of Hoare triples, we need a set of inference rules that tells us how to manipulate them. We write $\vdash \{\varphi\}\ c\ \{\varphi\}$ to indicate that we can prove the Hoare triple using these proof rules.

Let us consider the proof rules for Hoare logic over Imp commands. Recall the grammar for Imp commands:

$$c \quad ::= \quad \mathsf{skip} \mid x := a \mid c_1\ ;\ c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c$$

The proof rules are:

$$\text{[H-Skip]}\ \frac{}{\vdash \{\varphi\}\ \mathsf{skip}\ \{\varphi\}} \qquad \text{[H-Assign]}\ \frac{}{\vdash \{\varphi[x \mapsto a]\}\ x := a\ \{\varphi\}}$$

$$\text{[H-Seq]}\ \frac{\vdash \{\varphi\}\ c_1\ \{\chi\} \qquad \vdash \{\chi\}\ c_2\ \{\psi\}}{\vdash \{\varphi\}\ c_1\ ;\ c_2\ \{\psi\}} \qquad \text{[H-If]}\ \frac{\vdash \{b \wedge \varphi\}\ c_1\ \{\psi\} \qquad \vdash \{\neg b \wedge \varphi\}\ c_2\ \{\psi\}}{\vdash \{\varphi\}\ \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{\psi\}}$$

$$\text{[H-While]}\ \frac{\vdash \{b \wedge \varphi\}\ c\ \{\varphi\}}{\vdash \{\varphi\}\ \mathsf{while}\ b\ \mathsf{do}\ c\ \{\varphi \wedge \neg b\}} \qquad \text{[H-Weaken]}\ \frac{\varphi \Rightarrow \varphi' \qquad \vdash \{\varphi'\}\ c\ \{\psi'\} \qquad \psi' \Rightarrow \psi}{\vdash \{\varphi\}\ c\ \{\psi\}}$$

There are a few interesting notes in these rules. H-Assign may at first appear to have the substitution on the wrong side, but it does not. The rule says that, given a formula φ, it initially holds when you replace every

instance of x with a—and therefore the result does not reference x—then it will hold, even with x present, after setting the value of x in memory to a.

H-While is notable, as it refers to only φ and b, and no second formula ψ. Because we do not know how many times the loop will run, we are left operating with (potentially complicated and conditional) *loop invariants*. In this case, φ serves as a loop invariant. Lastly, H-Weaken (sometimes called the "rule of consequence") appeals to logical implication in the underlying logic. We are allowed to strengthen the precondition and weaken the postcondition arbitrarily, thus producing a weaker partial correctness statement.

2.1.1 Example Revisited

How might we prove that, given the example program in Section 1.1, the desired conditions hold. That is, if we consider call the full program c, how do we prove $\vdash \{p \geq 0\}\ c\ \{y = x^p\}$?

The first thing to note is that the H-Weaken rule allows us to arbitrarily change the structure of our conditions as long as they follow the required implications. In particular, we can note that $1 = x^0$ is always true and add it to the precondition, and rewrite the postcondition as $y = x^i \wedge p \geq i \wedge \neg(i < p)$. That is,

$$\text{H-Weaken}\ \frac{\vdash \left\{p \geq 0 \wedge 1 = x^0\right\}\ c\ \left\{y = x^i \wedge p \geq i \wedge \neg(i < p)\right\}}{\vdash \{p \geq 0\}\ c\ \{y = x^p\}}$$

We can now split the command into two sub-commands: the initializing assignments, and the while loop. Using H-Seq, we can construct the intermediate condition $\chi = p \geq i \wedge y = x^i$. That is, we need to prove

$$\vdash \left\{p \geq 0 \wedge 1 = x^0\right\}\ y := 1\ ;\ i := 0\ \left\{p \geq i \wedge y = x^i\right\}$$

and

$$\vdash \left\{p \geq i \wedge y = x^i\right\}\ \begin{array}{l}\mathsf{while}\ (i < p)\\ \mathsf{do}\ y := y * x\ ;\\ \quad i := i + 1\end{array}\ \left\{y = x^i \wedge p \geq i \wedge \neg(i < p)\right\}$$

The first is straightforward using H-Seq and H-Assign. For the second, this is exactly the structure we need to apply H-While, so the derivation reduces to proving

$$\vdash \left\{p \geq i \wedge y = x^i \wedge i < p\right\}\ y := y * x\ ;\ i := i + 1\ \left\{p \geq i \wedge y = x^i\right\}.$$

To prove this, we need to again transform the precondition, noting that the state precondition is equivalent to $y * x = x^{i+1} \wedge i + 1 \leq p$. From here, using H-Seq and H-Assign completes the proof.

2.2 Semantic Correctness

To define the semantic validity of a PCA, and therefore Hoare logic, we will use the notation $\vDash \{\varphi\}\ c\ \{\psi\}$. We also use the notation $\sigma \vDash \varphi$ to mean that assertion φ is true in state σ. Notice the double line on the turnstyle, which is often used for a semantic (or dynamic) proof, compared to the single line above, used for static proof.

These notations allow us to define semantic validity of a PCA in terms of any one of the semantics we have already defined. For simplicity, we will use the denotational semantics from the previous lecture.

$$\vDash \{\varphi\}\ c\ \{\psi\} \quad \stackrel{\triangle}{\iff} \quad \forall \sigma.\, \sigma \vDash \varphi \implies (C[\![c]\!]\,\sigma) \vDash \psi.$$

This definition makes sense for terminating executions, but we run into a slight hitch for nonterminating ones. Recall that $C[\![c]\!]\,\sigma = \bot$ whenever c diverges with input state σ. We could consider only cases where $C[\![c]\!]\,\sigma \neq \bot$, but that would be awkward to work with. Instead, we simply define $\bot \vDash \psi$ to hold for all ψ.[1] Among other things, that means a nonterminating program satisfies all postconditions, so $\vDash \{\varphi\}\ c\ \{\mathit{false}\}$ if and only if c always diverges on inputs that satisfy φ—that is, $\sigma \vDash \varphi \implies C[\![c]\!]\,\sigma = \bot$.

[1] Note that this approach is only appropriate for *partial* correctness assertions. For *total* correctness we would want the opposite requirement: that $\bot \nvDash \psi$ for all ψ.

2.3 Soundness and Completeness

A deduction system defines what it means for a formula to be *provable*, whereas a semantics defines what it means for a formula to be *true*. Given a logic with a semantics and a deduction system, two desirable properties are soundness and completeness.

- **Soundness:** The deduction system is *sound* if every provable statement is true.
- **Completeness:** The deduction system is *complete* if every true statement is provable.

Note that both soundness and completeness are relative terms. A logic may be sound (or complete) with respect to one semantics and not with respect to another.

Soundness is a basic property of a useful logical system. A logic with false theorems would not be very useful! With respect to our three existing semantics for Imp, the Hoare logic present above is sound.

Theorem 1 (Soundness of Hoare Logic)**.** *For any conditions φ and ψ and any program c,*

$$\vdash \{\varphi\}\ c\ \{\psi\} \implies \vDash \{\varphi\}\ c\ \{\psi\}.$$

Proof. This proof follows by structural induction on the derivation of $\vdash \{\varphi\}\ c\ \{\psi\}$ with careful use of the denotational semantics $C[\![\cdot]\!]$. The details are left as an exercise. □

Completeness, however, is more complicated. Hoare logic, as presented, is not complete in general. For instance, consider the PCA $\{true\}\ c\ \{false\}$. This assertion is semantically valid if and only if c diverges on all input states. A proof system that were sound and complete would thus provide a computable way of checking if c diverges on all inputs—one standard formulation of the halting problem.

Hoare logic is, however, *relatively complete*, relative to the truth of the underlying logic. That is, if the underlying logic is expressive enough, then given an oracle that can check the truth of statements in the underlying logic, one can prove any true PCA using Hoare logic. This is a famous result due to Stephen Cook (b. 1939), who also discovered NP-completeness. Although first-order logic is not expressive enough to provide relative completeness over arbitrary domains of computation, it is expressive enough over $\mathbb{N}$ or $\mathbb{Z}$. As a result, Hoare logic is relatively complete for Imp programs over integers. The notion of "expressive enough" for the underlying logic is that it must be always be able to express something called the *weakest precondition* that we discuss below. The following is the formal theorem statement.

Theorem 2 (Relative Completeness)**.** *Given a logical system L that is sufficient to express $WP(c, \psi)$ for any command c and predicate ψ, if all true statements in L are taken to be axioms, then*

$$\vDash \{\varphi\}\ c\ \{\psi\} \implies \vdash \{\varphi\}\ c\ \{\psi\}.$$

3 Weakest Preconditions

Given a postcondition ψ and a program c, one may wish to know: what precondition is required such that ψ will always be satisfied after executing c? This precondition is known as the *weakest precondition* (or sometimes the *weakest liberal precondition* since c is not required to terminate). That is, given a command c and postcondition ψ, the weakest precondition φ, which we will write $WP(c, \psi)$, is the logically weakest condition such that $\vDash \{\varphi\}\ c\ \{\psi\}$ holds. Here "weakest" means that any other valid precondition implies φ, formally defined as follows.

Definition 1. We say a formula is the *weakest precondition* of c and ψ, denoted $WP(c, \psi)$ if

$$\forall \sigma \in \textbf{Store}.\ \sigma \vDash WP(c, \psi) \iff (C[\![c]\!]\ \sigma) \vDash \psi.$$

Notice how this relates to partial correctness assertions in general. For any valid precondition φ, it must be the case that φ is at least as strong as $WP(c, \psi)$. To see why, we unfold some definitions.

$$\begin{aligned} \vDash \{\varphi\}\ c\ \{\psi\} \quad &\iff \quad \forall \sigma.\, \sigma \vDash \varphi \implies (C[\![c]\!]\,\sigma) \vDash \psi \\ &\implies \quad \forall \sigma.\, \sigma \vDash \varphi \implies \sigma \vDash WP(c, \psi). \end{aligned}$$

This last implication corresponds precisely to $\varphi \Rightarrow WP(c, \psi)$. Since $WP(c, \psi)$ is a valid precondition for c and ψ—that is, $\forall c, \psi.\, \vDash \{WP(c, \psi)\}\ c\ \{\psi\}$—we can combine it with modus ponens to show

$$\forall c, \varphi, \psi.\, \vDash \{\varphi\}\ c\ \{\psi\} \iff (\varphi \Rightarrow WP(c, \psi)). \tag{1}$$

3.1 Predicate Transformers

We may wish to compute $WP(c, \psi)$, and for Imp this is possible. The function that computes it, which we will denote $\mathsf{wp}(c, \psi)$, is what is known as a *predicate transformer* because it transforms one predicate into another—in this case a postcondition into its weakest precondition.

We can compute $\mathsf{wp}(c, \psi)$ for most of Imp similarly to how we constructed the denotational semantics.

$$\begin{aligned} \mathsf{wp}(\mathsf{skip}, \psi) &\triangleq \psi \\ \mathsf{wp}(x := a, \psi) &\triangleq \psi[x \mapsto a] \\ \mathsf{wp}(c_1\ ;\ c_2, \psi) &\triangleq \mathsf{wp}(c_1, \mathsf{wp}(c_2, \psi)) \\ \mathsf{wp}(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \psi) &\triangleq (b \Rightarrow \mathsf{wp}(c_1, \psi)) \wedge (\neg b \Rightarrow \mathsf{wp}(c_2, \psi)) \end{aligned}$$

Again, while loops pose a challenge. We would like a condition that satisfies the following equation:

$$\mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c, \psi) = (\neg b \Rightarrow \psi) \wedge (b \Rightarrow \mathsf{wp}(c, \mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c, \psi))).$$

As with the denotational semantics, this would not be a valid definition because it is circular. We can again build up a condition using a sequence of approximations.

$$\begin{aligned} F_0(\psi) &= \mathit{true} \\ F_{i+1}(\psi) &= (\neg b \Rightarrow \psi) \wedge (b \Rightarrow \mathsf{wp}(c, F_i(\psi))) \end{aligned}$$

The limit of this sequence is the conjunction of all of the elements, so we can define

$$\mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c, \psi) \triangleq \bigwedge_{i=0}^{\infty} F_i(\psi).$$

It is possible to encode $\mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c, \psi)$ as an ordinary assertion. See Chapter 7 of "The Formal Semantics of Programming Languages: An Introduction" by Glynn Winskel for details.

To check that this definition is correct, we need to prove that it yields a valid weakest precondition. That is, $\mathsf{wp}(c, \psi) = WP(c, \psi)$. The proof is left as an exercise.

3.2 Relative Completeness

This definition of the weakest precondition fills out the definitions in the statement of relative completeness (Theorem 2), meaning the theorem statement is now well-defined. To prove it for Imp, we need one more lemma.

Lemma 1. *For any command c and predicate ψ,*

$$\vdash \{\mathsf{wp}(c, \psi)\}\ c\ \{\psi\}\,.$$

Proof. By induction on c. Only the while case is interesting.

Let $c = \mathsf{while}\ b\ \mathsf{do}\ c_0$ and $\varphi = \mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c_0, \psi)$. By construction,

$$\varphi = (\neg b \Rightarrow \psi) \wedge (b \Rightarrow \mathsf{wp}(c_0, \varphi)),$$

so therefore $b \wedge \varphi$ simplifies to $\mathsf{wp}(c_0, \varphi)$.

The inductive hypothesis immediately proves $\vdash \{\mathsf{wp}(c_0, \varphi)\}\ c\ \{\varphi\}$, which is, equivalently $\vdash \{b \wedge \varphi\}\ c\ \{\varphi\}$. The H-WHILE rule then proves $\vdash \{\varphi\}\ c\ \{\varphi \wedge \neg b\}$. However, if we expand out this postcondition,

$$\neg b \wedge \varphi = \neg b \wedge (\neg b \Rightarrow \psi) \wedge (b \Rightarrow \mathsf{wp}(c_0, \varphi)) = \psi.$$

Therefore, $\vdash \{\mathsf{wp}(\mathsf{while}\ b\ \mathsf{do}\ c_0, \psi)\}\ \mathsf{while}\ b\ \mathsf{do}\ c_0\ \{\psi\}$, completing the case. □

This lemma coupled with equation (1) are sufficient to prove Theorem 2 for IMP.

Theorem 2 (Relative Completeness)**.** Given a logical system L that is sufficient to express $\mathsf{wp}(c, \psi)$ for any command c and predicate ψ, if all true statements in L are taken to be axioms, then

$$\vDash \{\varphi\}\ c\ \{\psi\} \implies \vdash \{\varphi\}\ c\ \{\psi\}.$$

Proof. Assume that $\vDash \{\varphi\}\ c\ \{\psi\}$. Given the predicate transformer above and the fact that $\mathsf{wp}(c, \psi)$ is, in fact, the weakest precondition of c and ψ, equation (1) gives $\varphi \Rightarrow \mathsf{wp}(c, \psi)$. Applying this result and the H-WEAKEN rule to the result of Lemma 1 immediately proves $\vdash \{\varphi\}\ c\ \{\psi\}$. □

4 Hoare Logic and Predicate Transformers as Semantics

To this point, we have defined Hoare logic and the wp predicate transformer and proved their soundness and relative completeness with respect to an existing denotational semantics. We could, however, take Hoare logic or wp to *be* the semantics for IMP.

If we did so, the description of how commands relate pre- and postconditions would become the definition of the meaning of IMP. Other semantics, such as the denotational semantics we reference above or one of the structured operational semantics discussed previously, could then be shown equivalent. Indeed, several of the results we proved about the relationship between PCAs and stores satisfying predicates would become critical parts of that proof.