1 Program verification

We have covered a number of general topics so far in the course: propositional logic, predicate logic, recursively defined sets, and modal logic. Each of these topics might also be covered in a course on mathematical logic and you may be wondering about the connection to computer science.

In fact, one of the most important—perhaps the single most important—applications of formal logic is to computer science and related areas like electrical engineering. You have already seen examples of using formal methods to solve computational problems in sports tournament scheduling and network security analysis. These applications were quite specific; here we will focus on applying formal methods to a much more general problem: *program verification*.

The goal of program verification is to prove that some program of interest has certain behavior. We have already seen some simple examples of this, such as when we used induction to prove that the addition function on naturals is commutative. That kind of proof is based on *operational semantics*—that is, the execution behavior of the code (*e.g.*, the add function). Proofs based on operational semantics are quite useful but hard to generalize to larger programs.

Our goal now is to introduce an alternative technique for program verification called *Hoare logic*. Hoare logic is a kind of modal logic that is specialized for reasoning about program verification. We proceed by example; first we explain a simple programming language and provide a model for computational state. Next, we define a modal logic of assertions and provide an *axiomatic semantics* for our language by defining series of *Hoare rules*. Finally, we show how to apply those rules to reason about a number of example programs.

2 Programming language

We start by defining a very simple programming language. The language is too brain-dead to make programming in it much fun, since it lacks lots of features that make a programmer's task easier (*e.g.*, function calls). Still, it is Turing complete—that is, it is capable of doing any computation that any other machine is capable of doing, if the programmer is willing to work hard enough.

Expressions. We start with the notions of numerical and boolean expressions:

$$\begin{array}{rrcl} \text{numerical expressions} & e & \equiv & z \mid x \mid e_1 \texttt{+} e_2 \mid e_1 \texttt{-} e_2 \mid e_1 \texttt{*} e_2 \\ \text{boolean expressions} & b & \equiv & e_1 \texttt{<=} e_2 \mid b_1 \texttt{||} b_2 \mid \texttt{!}\, b \end{array}$$

Numerical expressions e are inductive types with two kinds of base elements: integers z or program variables x. The recursive cases of e are for the sum, difference, and product of two other numerical expressions e_1 and e_2, respectively. Boolean expressions have only one kind of base element, the comparison of two numerical expressions e_1 and e_2; in addition they have two recursive cases, for disjunction and negation. Notice that we are missing certain other

obvious boolean expressions; it is convenient to define the following macros:

$$\begin{array}{lcl}
\texttt{true} & \equiv & \texttt{0<=0} \\
\texttt{false} & \equiv & \texttt{!true} \\
b_1\texttt{\&\&}b_2 & \equiv & \texttt{!((!}b_1\texttt{)||(!}b_2\texttt{))} \\
e_1\texttt{==}e_2 & \equiv & (e_1\texttt{<=}e_2)\texttt{\&\&}(e_2\texttt{<=}e_1) \\
e_1\texttt{!=}e_2 & \equiv & \texttt{!(}e_2\texttt{==}e_2\texttt{)} \\
e_1\texttt{<}e_2 & \equiv & (e_1\texttt{<=}e_2)\texttt{\&\&}(e_1\texttt{!=}e_2)
\end{array}$$

These macros are as you would expect; notice that conjunction uses deMorgan.

Expression evaluation. Once we have an expression, the natural thing to do is to evaluate it into some (numerical or boolean) value. For simple numerical expressions like the following, this is not so hard:

$$\texttt{3+(2*(0-2))}$$

Clearly this expression evaluates to -1. The tricky point is that numerical expressions can contain program variables, like this one (let's call it e_a):

$$e_a \quad \equiv \quad \texttt{m*(2+a)}$$

Here `m` and `a` are two program variables. The value of the expression e_a clearly depends on the values of `m` and `a`: for example, if `m` $= 2$ and `a` $= 3$, then the expression e_a evaluates to 10. To help, we introduce the notion of a *context*:

$$\text{context} \quad \rho \quad \equiv \quad \mathbb{X} \to \mathbb{Z}$$

That is, a context ρ is a function[1] from the set from the set of variable names $\mathbb{X}$ to the set of integers $\mathbb{Z}$. The context above (we'll call it ρ_a) is defined as:

$$\rho_a(x) \quad \equiv \quad \begin{cases} 2 & \text{when } x = \texttt{m} \\ 3 & \text{when } x = \texttt{a} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given this notion of context, it is easy to define the numeric evaluator function $\mathsf{neval}(\rho, e)$ that evaluates e in the context ρ by:

$$\mathsf{neval}(\rho, e) \quad \equiv \quad \begin{cases} z & \text{when } e = z \\ \rho(x) & \text{when } e = x \\ \mathsf{neval}(\rho, e_1) + \mathsf{neval}(\rho, e_2) & \text{when } e = e_1\texttt{+}e_2 \\ \mathsf{neval}(\rho, e_1) - \mathsf{neval}(\rho, e_2) & \text{when } e = e_1\texttt{-}e_2 \\ \mathsf{neval}(\rho, e_1) \times \mathsf{neval}(\rho, e_2) & \text{when } e = e_1\texttt{*}e_2 \end{cases}$$

Hopefully our "overloading" of the arithmetical symbols is clear. When we write $\mathsf{neval}(\rho, e_1) + \mathsf{neval}(\rho, e_2)$ we mean to use the standard addition function

[1] We use the symbol "$\to$" here to indicate the set of functions.

on integers, whereas when we write $e = e_1$+e_2 we are indicating one of the constructors of the inductively defined set of numerical expressions. Although usually it is not hard to understand which symbol we mean in a given context, we also use different fonts to help distinguish which meaning is indicated. Using our previously defined expression e_a and context ρ_a, it should be clear that

$$\mathsf{neval}(\rho_a, e_a) \quad = \quad 10$$

Boolean expressions are evaluated in much the same way; evaluation here also needs a context so that numerical comparisons can be properly evaluated:

$$\mathsf{beval}(\rho, b) \quad \equiv \quad \begin{cases} \mathsf{neval}(\rho, e_1) \leq \mathsf{neval}(\rho, e_2) & \text{when } b = e_1 \texttt{<=} e_2 \\ \mathsf{beval}(\rho, b_1) \vee \mathsf{beval}(\rho, b_2) & \text{when } b = b_1 \texttt{||} b_2 \\ \neg \mathsf{beval}(\rho, b') & \text{when } b = \texttt{!}b' \end{cases}$$

Hopefully our notation here is straightforward. The beval function turns boolean expressions into truth values like $\top$ and $\bot$.

Commands. Now that we have defined the expressions (and their evaluation) in our language, we are ready to define its commands. Here they are:

Syntax	Command	Description
`skip`	skip	Do nothing; nop
x `:=` e	assign	Store e into the variable x
c_1 `;` c_2	sequence	Run c_1 followed by c_2
`if (`b`)` $\{c_1\}$ `else` $\{c_2\}$	if	If b is `true` then run c_1 else run c_2
`while (`b`)` $\{c\}$	loop	As long as b is `true` then run c

We have only five commands; the result is an imperative language like C or Java, but hugely simplified. One (very useful) way to think about commands is that they transform a given context ρ_1 into a succeeding context ρ_2.

The first command, `skip`, has no effect—the succeeding context is equal to the starting context. The second command, x `:=` e (assignment), evaluates the expression e and stores the result into the program variable x—the succeeding context has a new value for variable x. The third command, c_1 `;` c_2 (sequence), runs the command c_1 and then runs the command c_2. The succeeding context of c_1 becomes the proceeding context of c_2, and the succeeding context of the entire sequence is the succeeding context of c_2. The fourth command, `if (`b`)` $\{c_1\}$ `else` $\{c_2\}$ (if/conditional), evaluates b; if the result is `true` then it runs c_1; otherwise it runs c_2. The succeeding context of the conditional is the succeeding context of whichever instruction ran. Finally, the fifth command, `while (`b`)` $\{c\}$ (while), evaluates b and if the result is `true` runs c, and then loops around (evaluates b again, which may have a different value after running c, and if b still evaluates to `true` then runs c again, before looping around again and testing $b\ldots$). All of these commands should be familiar to you (except maybe `skip`; if you want you can substitute the assignment x `:=` x).

Example 1. To illustrate our language, consider this simple program:

```
y := x;
while (2<=y)
{
  y := y-2
}
```

What does this program (let us call it "Parity") do? Parity first copies the value of `x` into `y` and then sets up a loop that keeps going until the program variable `y` is less than 2; until this is the case, it keeps decrementing `y` by 2. The intention of the Parity program is that if the program variable `x` starts as an even number, then when the program ends `y` will be equal to 0. On the other hand, when `x` starts as an odd number, then when the program ends `y` will be equal to 1. Thus, Parity seems to determine if a number is even or odd.

Example 2. Consider also the slightly more complicated "Factorial" program:

```
y := 1;
z := 0;
while (z!=x)
{
  z := z+1;
  y := y*z
}
```

Here we start by setting the program variables `y` and `z` to 1 and 0, respectively. We then enter a loop where we increment `z` and multiply the value of `y` by the new value of `z`; we exit the loop when `z==x`. The purpose of the Factorial program is to compute `x`! into `y`—*i.e.*, after the program runs we should have

$$\texttt{y} \quad = \quad 1 \times 2 \times \ldots \times (\texttt{x} - 1) \times \texttt{x}$$

Exercise 1. (⋆) Prove that our language is Turing complete. The major challenge is representing an infinite tape with a finite number of named variables.

3 Hoare Logic

Now that we have an understanding of our programming language, we turn to the problem of program verification. The first question must be, what do we mean by "proving that a program has certain behavior"? This is actually a pretty complicated question. For example, quicksort and mergesort are two standard sorting algorithms, although they produce the same result given the same input, they are not the same algorithm (*e.g.*, they may use different amounts of memory, or have different running times).

We have to determine which properties of programs we are interested in verifying. The most fundamental property of a program is neither its running time nor its memory requirements, but rather whether it produces the output you expect for a given input. Another way of saying that is, given some starting context ρ_α that satisfies some initial property P_α, once the program terminates, the ending context ρ_ω satisfies some final property P_ω. For example, in the Parity program, we might like to verify that given any starting context, the ending context must have `y` equal to 0 if `x` is even, and `y` equal to 1 if `x` is odd.

Assertions. You may be noticing a connection to previous material. The truth or falsehood of a property **depends on the current context**. In other words, we have a modal logic in which our set of worlds are our program contexts. It is entirely reasonable to borrow notation used in modal logic as well and write:

$$\rho_\alpha \Vdash P_\alpha$$

to say that the initial property P_α holds in the initial context ρ_α, and

$$\rho_\omega \Vdash P_\omega$$

to say that the final property P_ω holds in the final context ρ_ω.

To define the logical operators in our modal logic we do our usual trick of lifting operators from the metalogic. When we did this for propositional and predicate logic, our metalogic was *English*, which was good because it provided intuition but troublesome because natural language is often ambiguous. Now that we understand propositional and predicate logic, however, they are perfectly reasonable candidates for our metalogic. We proceed as follows:

ρ	$\Vdash$	$\top$	$\equiv$	$\top$	truth constant
ρ	$\Vdash$	$\bot$	$\equiv$	$\bot$	falsehood constant
ρ	$\Vdash$	$P \wedge Q$	$\equiv$	$(\rho \Vdash P) \wedge (\rho \Vdash Q)$	conjunction
ρ	$\Vdash$	$P \vee Q$	$\equiv$	$(\rho \Vdash P) \vee (\rho \Vdash Q)$	disjunction
ρ	$\Vdash$	$P \to Q$	$\equiv$	$(\rho \Vdash P) \to (\rho \Vdash Q)$	implication
		$\neg P$	$\equiv$	$P \to \bot$	negation

On the left-hand side we give the new operators of our modal logic, and on the right-hand side we provide their definitions using the operators of our metalogic (predicate logic)[2]. To avoid massive symbol overload we use the same symbols on both sides of the definitions but of course they are not the same.

We also want to have universal $\forall$ and existential $\exists$ quantifiers in our modal logic. Fortunately, these too can be lifted straight from our metalogic:

ρ	$\Vdash$	$\forall x\ P$	$\equiv$	$\forall x\ (\rho \Vdash P)$	universal
ρ	$\Vdash$	$\exists x\ P$	$\equiv$	$\exists x\ (\rho \Vdash P)$	existential

Here we assume that x is free in P so that the quantifiers can provide meaning.

[2]This is exactly the same technique that was used in the most recent Coq homework and you may find it useful to review how it was done there.

It is often very useful to embed predicate logic formulas into modal formulas. We define the *plain lift* operator $\langle\cdot\rangle$ to do just that:

$$\rho \quad \Vdash \quad \langle P \rangle \qquad \equiv \qquad P \quad \Big| \text{ plain lift}$$

That is, $\langle P \rangle$ holds in any context ρ if and only if P is true in predicate logic.

All of the operators defined so far are either independent of the context ρ ($\top$, $\bot$, $\langle\cdot\rangle$) or simply "pass on" the given context to their subformulas. What about formulas whose truth critically depends on the context? The simplest of these allow us to lift numeric and boolean expressions into our assertion logic:

$$\begin{array}{llll} \rho & \Vdash & e \Downarrow z & \equiv \quad \mathsf{neval}(\rho, e) = z \quad \Big| \text{ numeric evaluation} \\ \rho & \Vdash & [b] & \equiv \quad \mathsf{beval}(\rho, b) = \top \quad \Big| \text{ boolean evaluation} \end{array}$$

When $e \Downarrow z$ holds on some context ρ, then the evaluation of e in ρ is equal to z; similarly, when $[b]$ holds on some context ρ, then b evaluates to $\top$ in ρ.

You probably remember that when we define a modal logic we usually define not only a set of worlds but also a relation between worlds. Here we will generalize this notion by allowing many different kinds of relations between worlds—in other words, we have a *multi*modal logic. One important family of world relations is the update relation $U_{(x,n)}$ defined as follows:

$$\rho \; U_{(x,n)} \; \rho' \qquad \equiv \qquad \rho' = [x \mapsto n]\rho$$

In other words, two contexts ρ and ρ' are related by $U_{(x,n)}$ if they are equal everywhere other than at variable x, and the value of ρ' at x is equal to n.

We can define the modal operator $\Box^U_{(x,n)}$ in the usual way by universally quantifying over reachable worlds via this relation:

$$\rho \Vdash \Box^U_{(x,n)} P \qquad \equiv \qquad \forall \rho' \; \big((\rho \; U_{(x,n)} \; \rho') \to (\rho' \Vdash P) \big) \qquad \Big| \text{ store update modality}$$

In other words, if $\Box^U_{(x,n)} P$ holds on some context ρ, then P would hold if ρ **were updated so that variable** x **had value** n. We can also define the related:

$$\rho \Vdash [x \mapsto e] P \qquad \equiv \qquad \rho \Vdash \Box^U_{(x,\mathsf{neval}(\rho,e))} P \; \Big| \text{ store expression update modality}$$

In this case, P would hold if ρ were updated so that variable x had the result of evaluating e in ρ. We will see how this modality is used shortly.

What about other kinds of relations between worlds? There is a second very important class of relations. Notice that **the commands of our language** move from one world to the next by computing the subsequent context from the proceeding one. For now we will avoid formal definitions of this modality, but we will return to it next week when we study the soundness of Hoare logic.

Hoare rules. Now that we have defined a modal logic of properties/assertions, we can return to the considering our goals for the verification. We are trying to find a method for proving that from a context that satisfies some initial property P_α, we reach a final context that satisfies some final property P_ω.

This is exactly the idea behind the *Hoare triple*. A Hoare triple, written

$$\{P\}\ c\ \{Q\}$$

is a way of saying that given any starting context that satisfies the property P, after running the command c, we will be in an ending context that satisfies the property Q. Of course, the truth of a particular Hoare tuple depends very much on the particular P, c, and Q! Some tuples seem pretty reasonable, such as:

$$\{\texttt{x} \Downarrow 5\}\ \texttt{x := x+1}\ \{\texttt{x} \Downarrow 6\}$$

That is, starting from any context in which the expression `x` evaluates to 5, after incrementing `x`, we will be in a context in which `x` evaluates to 6. In contrast, the following tuple does not seem so reasonable:

$$\{\texttt{x} \Downarrow 5\}\ \texttt{x := x+1}\ \{\bot\}$$

In other words, this Hoare tuple is claiming that, given a starting context in which `x` evaluates to 5, by incrementing `x` we reach no context at all (that is, the increment command "blows up"—or maybe increment never terminates). This seems pretty unlikely, since a reasonable intuition about the action of the increment command indicates that such a context should exist.

Unfortunately, we are not quite in a position to prove that the second Hoare tuple is invalid, since we have not pinned down the formal behavior of our commands, or given a formal definition for Hoare tuples. These important topics are the focus of next week's lecture. Until then, we will proceed axiomatically: first, we give a simple series of rules (axioms) justified by our intuition; second, we show how to combine those axioms into proofs about whole programs.

Our first axiom is for `skip`; since `skip` is very simple (do nothing!), you would expect that it would have a very simple axiom, and it does:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \textsf{Skip}$$

In other words, the Skip axiom says that for any starting property P, given an initial context ρ satisfying P, after running the command `skip` (which does nothing), we reach a concluding context (which is just ρ again) that satisfies P.

The axiom for x `:=` e (assignment) uses the store expression update modality:

$$\frac{}{\{[x \mapsto e]\ P\}\ x\ \texttt{:=}\ e\ \{P\}}\ \textsf{Assignment}$$

In other words, supposing that some property P would hold if the initial context were updated so that variable `x` mapped to (the evaluated) expression e. Since the assignment command does exactly that—transform the initial context into a new context in which `x` now contains the (evaluated) expression e, it makes perfect sense that P would hold in the concluding context.

You might notice that our Hoare axiom Assignment does not quite fit the pattern of our initial (reasonable) example of a Hoare triple above. To validate

that program, we will need another Hoare axiom called Consequence:

$$\frac{P \to P' \qquad Q' \to Q \qquad \{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ \textsf{Consequence}$$

In other worlds, suppose we have proved some Hoare tuple $\{P'\}\ c\ \{Q'\}$, and further suppose that we have some (possibly) stronger precondition P and some (possibly) weaker postcondition Q. In that case, we are justified in concluding $\{P\}\ c\ \{Q\}$: if we know that our initial satisfies the (stronger) P, we know that it must also satisfy P'; thus we know that the concluding state must satisfy the (stronger) Q', which implies that it also satisfies Q. Let's see this in action:

$$\frac{\dfrac{(\mathtt{x} \Downarrow 5) \to \big([\mathtt{x} \mapsto \mathtt{x{+}1}](\mathtt{x} \Downarrow 6)\big) \qquad (\mathtt{x} \Downarrow 6) \to (\mathtt{x} \Downarrow 6)}{\{[\mathtt{x} \mapsto \mathtt{x{+}1}](\mathtt{x} \Downarrow 6)\}\ \mathtt{x := x{+}1}\ \{\mathtt{x} \Downarrow 6\}}\ \textsf{Assignment}}{\{\mathtt{x} \Downarrow 5\}\ \mathtt{x := x{+}1}\ \{\mathtt{x} \Downarrow 6\}}\ \textsf{Consequence}$$

We have successfully verified the desired specification (precondition/postcondition) pair. Notice that in order to use the Conqsequence rule we had to:

1. Prove that our preferred precondition $(\mathtt{x} \Downarrow 5)$ is stronger than the precondition required by Assignment ($[\mathtt{x} \mapsto \mathtt{x{+}1}](\mathtt{x} \Downarrow 6)$). This proof is not contained in the above derivations, but fortunately it is not very difficult. If we are in a context where x has value 5, then if we move to a new context in which x contains the value is one greater than the old value of x—then we will be in a context in which x has value 6.

2. Prove that our desired postcondition is implied by the postcondition we get from the implication rule; since they are the same that is very simple.

3. Apply the Assignment rule to get the underlying Hoare triple.

All of the previously rules only apply to program fragments with a single command. One of the "big wins" of Hoare logic is that the rules can be composed in a modular way. The first composition rule is Sequence:

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\,;\ c_2\ \{R\}}\ \textsf{Sequence}$$

That is, from specifications of the subcommands c_1 and c_2, we can derive specifications for the combined sequence $c_1\,;\ c_2$. We apply Consequence if the obvious postcondition for c_1 does not exactly match the obvious precondition for c_2. Let's take a look at an example of verifying a sequence[3]:

$$\frac{\dfrac{\ldots\text{exactly as above}\ldots}{\{\mathtt{x} \Downarrow 5\}\ \mathtt{x := x{+}1}\ \{\mathtt{x} \Downarrow 6\}}\ \textsf{Conseq.} \qquad \dfrac{\dfrac{\ldots\text{similar to above}\ldots}{\{\mathtt{x} > 5\}\ \mathtt{x := x{+}1}\ \{\mathtt{x} > 6\}}\ \textsf{Conseq.}}{\{\mathtt{x} \Downarrow 6\}\ \mathtt{x := x{+}1}\ \{\mathtt{x} > 6\}}}{\{\mathtt{x} \Downarrow 5\}\ \mathtt{x := x{+}1\,;\ x := x{+}1}\ \{\mathtt{x} > 6\}}\ \begin{array}{l}\textsf{Conseq.}\\ \textsf{Sequence}\end{array}$$

[3]Here we write, *e.g.*, $\mathtt{x} > 6$ as shorthand for $\exists n\ \big((x \Downarrow n) \wedge \langle n > 6 \rangle\big)$.

To avoid running into the margins in the proceeding derivation we omitted the following side conditions of the lower-right application of Consequence:

$$(\mathtt{x} \Downarrow 6) \to (\mathtt{x} > 5) \qquad \text{and} \qquad (\mathtt{x} > 6) \to (\mathtt{x} > 6)$$

Of course, these are very easy. Still, it is easy to observe that our derivations are growing rapidly. We need to introduce a better notation to handle the problem. The idea is that we will write the assertions in between the lines of the program, applying the natural rules to get us from on assertion to the next. For example, here is the same verification in the new format:

$\{\mathtt{x} \Downarrow 5\}$
$\{[\mathtt{x} \mapsto \mathtt{x+1}](\mathtt{x} \Downarrow 6)\}$
`x := x+1;`
$\{\mathtt{x} \Downarrow 6\}$
$\{\mathtt{x} > 5\}$
$\{[\mathtt{x} \mapsto \mathtt{x+1}](\mathtt{x} > 6)\}$
`x := x+1`
$\{\mathtt{x} > 6\}$

Clearly this kind of format is more compact than full proof-tree derivations. In fact, it is (usually) not very hard to decode what is going on, since there are only two possibilities for getting from assertion to the next.

In the first case, applicable whenever there are two assertions in a row, the rule of consequence has been applied to get from one assertion to the next. If there are more than two assertions in a row, then consequence has been applied repeatedly, usually because the verifier wants to provide additional clarity in his proof. For example, the first two lines of the verification above are assertions, and to get from the first to the second we must apply Consequence. This means that a side proof, usually omitted since it is usually very easy, must show that the preceding assertion must imply the succeeding assertion.

In the second case, applicable whenever two assertions are separated by a single command, the underlying proof rule must be applied. Here, since both commands are assignments, the Assignment rule has been applied. The Sequence rule is implied by the way the notation proceeds from one command to the next.

There are only a few axioms remaining. Let us examine the rule for `if`:

$$\frac{\{P \wedge [b]\}\ c_1\ \{Q\} \qquad \{P \wedge \neg[b]\}\ c_2\ \{Q\}}{\{P\}\ \mathtt{if}\ (b)\ \{c_1\}\ \mathtt{else}\ \{c_2\}\ \{Q\}}\ \mathsf{If}$$

This rule is somewhat similar to the disjunction-elimination rule from propositional logic. That is, if we can show that starting from a context that satisfies P, then running either c_1 or c_2 leaves us in a context that satisfies Q, then we can conclude that the entire conditional (if statement) has postcondition Q. The only wrinkle is that while verifying c_1 we are allowed to assume that the boolean was true ($[b]$) when verifying the postcondition Q; while verifying c_2 we are allowed to assume that the boolean was not true ($\neg[b]$).

To demonstrate the If rule, consider the following verification:

$\{(\texttt{x} \Downarrow 0) \vee (\texttt{x} \Downarrow 1)\}$
`if (x==0)`
`{`
  $\{\big((\texttt{x} \Downarrow 0) \vee (\texttt{x} \Downarrow 1)\big) \wedge [\texttt{x==0}]\}$
  $\{\texttt{x} \Downarrow 0\}$
  $\{[\texttt{x} \mapsto \texttt{x+2}](\texttt{x} \Downarrow 2)\}$
  `x := x+2`
  $\{\texttt{x} \Downarrow 2\}$
`} else {`
  $\{\big((\texttt{x} \Downarrow 0) \vee (\texttt{x} \Downarrow 1)\big) \wedge \neg[\texttt{x==0}]\}$
  $\{\texttt{x} \Downarrow 1\}$
  $\{[\texttt{x} \mapsto \texttt{x+1}](\texttt{x} \Downarrow 2)\}$
  `x := x+1`
  $\{\texttt{x} \Downarrow 2\}$
`}`
$\{\texttt{x} \Downarrow 2\}$

Notice we carefully distributed the precondition (plus $[b]/\neg[b]$) to both branches of the `if`, and ensured that both branches reached the same postcondition.

Finally, let us consider the `while` command. Here we have some decisions to make. Let us first present the following (perfectly reasonable) rule:

$$\frac{\{P \wedge [b]\}\ c\ \{P\}}{\{P\}\ \texttt{while (}b\texttt{)}\ \{c\}\ \{P \wedge [\neg b]\}}\ \textsf{While}$$

Suppose the loop body c can be verified with precondition P (plus the fact that the boolean b evaluated to true) and postcondition P. In that case, it is reasonable to conclude that if we start the `while` from precondition P, then if/when it finally terminates, it will end with postcondition P plus the fact that the boolean b finally evaluated to false. Here is why: the loop body cannot take us "out of" P; in other words, P is an *invariant* of the loop. Therefore, no matter how many times we go around the loop, P will remain true.

This raises all kinds of tricky questions. The first is, how do we discover the "right" loop invariant P for a given situation? In general, this is a very hard problem and is an active research problem decades after Hoare logic was developed. We suggest an iterative approach: try some invariant that you think might work and perform the verification. If the Hoare proof breaks somewhere, then take a guess at how the invariant might be changed to fix the break, and start the verification over. Continue until you have found something that works.

To see this rule in action, let's try to verify the Parity function with precondition $\top$ (make no restrictions on the starting context) and postcondition

$$\big((\texttt{y} \Downarrow 0) \wedge \textsf{even}(\texttt{x})\big) \vee \big((\texttt{y} \Downarrow 1) \wedge \textsf{odd}(\texttt{x})\big)$$

Note that to avoid large formulas, we will start to use abbreviations like $\texttt{x} = \texttt{y}$

$\{\top\}$
$\{[\mathtt{y} \mapsto \mathtt{x}](\mathtt{x} = \mathtt{y})\}$
`y := x;`
$\{\mathtt{x} = \mathtt{y}\}$
$\{\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\}$
`while (2<=y)`
{
$\quad\{\Big(\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\Big) \wedge [\mathtt{2<=y}]\}$
$\quad\{\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\}$
$\quad\{[\mathtt{y} \mapsto \mathtt{y\text{-}2}]\Big(\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\Big)\}$
$\quad$`y := y-2`
$\quad\{\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\}$
}
$\{\Big(\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\Big) \wedge \neg[\mathtt{2<=y}]\}$
$\{\Big(\exists n\ \big(\mathtt{x}\ =\ \mathtt{y} + (2 \times n)\big)\Big) \wedge (\mathtt{y} < 2)\}$
$\{\ldots \text{uh oh} \ldots\}$
$\{\big((\mathtt{y} \Downarrow 0) \wedge \mathsf{even}(\mathtt{x})\big) \vee \big((\mathtt{y} \Downarrow 1) \wedge \mathsf{odd}(\mathtt{x})\big)\}$

Figure 1: Attempted verification for Parity

instead of $\exists n\ (\mathtt{x} \Downarrow n \wedge \mathtt{y} \Downarrow n)$, $\mathsf{even}(\mathtt{x})$ instead of $\exists n\ \big(\mathtt{x} \Downarrow n \wedge \exists m\ \langle m{+}m = n\rangle\big)$, and so forth. We give the attempted verification in Figure 1.

The verification goes through just fine until the very end, when we reach the "uh oh". The problem is that the preceding assertion **does not imply** the desired postcondition. A little examination leads to the reason: `y` is an integer; what happens if `y` is **negative**? In that case the final conclusion will be false, since `y` will neither equal 0 or 1. It turns out that `y` will be negative if the initial value of `x` is negative. This is exactly the kind of mistake that is very easy to make when thinking about a program informally, and exactly the kind of mistake that a formal technique like Hoare logic will help you find.

We need to fix our specification by strengthening the precondition (*e.g.*, to $\mathtt{x} \geq 0$) or weakening the postcondition (*e.g.*, to $\top$). Here the right choice is to strengthen the precondition, yielding the following flawless given in Figure 2. We have put boxes around the formulas that are different in the revised proof; these track the fact that `y` is nonnegative during the program's execution. At the penultimate step we are able to combine the fact that `y` is nonnegative and the fact that $\mathtt{y} < 2$ to conclude that `y` is either 0 or 1. Note we carefully ensure we reach the loop invariant at the end of the loop body.

Verifying Factorial. To give another example, let us reconsider the Factorial program. We would like to show that given any starting context, after the

$\{\boxed{\mathtt{x} \geq 0}\}$

$\{[y \mapsto x](\mathtt{x} = \mathtt{y} \ \wedge \ \boxed{\mathtt{y} \geq 0})\}$

`y := x;`

$\{\mathtt{x} = \mathtt{y} \ \wedge \ \boxed{\mathtt{y} \geq 0}\}$

$\{\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n) \ \wedge \ \boxed{\mathtt{y} \geq 0})\}$

`while (2<=y)`

{

$\{\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 0}\Big) \wedge [\mathtt{2<=y}]\}$

$\{\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 2}\}$

$\{[\mathtt{y} \mapsto \mathtt{y-2}]\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 0}\Big)\}$

`y := y-2`

$\{\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 0}\}$

}

$\{\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 0}\Big) \wedge \neg[\mathtt{2<=y}]\}$

$\{\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \boxed{\mathtt{y} \geq 0}\Big) \wedge (\mathtt{y} < 2)\}$

$\{\big((\mathtt{y} \Downarrow 0) \wedge \mathsf{even}(\mathtt{x})\big) \vee \big((\mathtt{y} \Downarrow 1) \wedge \mathsf{odd}(\mathtt{x})\big)\}$

Figure 2: Correct verification for Parity

program terminates the value of y will be equal to the value of x!—that is,

$$\mathtt{y} \ = \ 1 \times 2 \times \ldots \times (\mathtt{x} - 1) \times \mathtt{x}$$

This is a little bit trickier to precisely specify than it might appear. The problem is that the factorial function is usually defined on **naturals**, but our program variables are **integers**. What do we mean when we say $\mathtt{y} = \mathtt{x}!$ if x is negative?

A little thought indicates several reasonable possibilities:

1. Define $z!$ to be -1 if $z < 0$
2. Define $z!$ to be equal to $(-z)!$ when $z < 0$ (that is, take the absolute value before calculating factorial)
3. Define $z!$ to be equal to $-((-z)!)$ when $z < 0$ (that is, take the absolute value before calculating factorial, then take the inverse)
4. Say that $z!$ is undefined when $z < 0$, and so when we say $\mathtt{y} = \mathtt{x}!$ we must include a side condition that $\mathtt{y} \geq 0$

$\{\top\}$
$\{[\mathtt{y} \mapsto 1](\mathtt{y} \Downarrow 1)\}$
$\mathtt{y} := 1;$
$\{\mathtt{y} \Downarrow 1\}$
$\{[\mathtt{z} \mapsto 0](\mathtt{y} \Downarrow 1 \ \wedge \ \mathtt{z} \Downarrow 0)\}$
$\mathtt{z} := 0;$
$\{\mathtt{y} \Downarrow 1 \ \wedge \ \mathtt{z} \Downarrow 0\}$
$\{\mathtt{y} = \mathtt{z}!\}$
`while (z!=x)`
$\{$
$\{(\mathtt{y} = \mathtt{z}!) \ \wedge \ [\mathtt{z}!=\mathtt{x}]\}$
$\{\big(\mathtt{y} \times (\mathtt{z}+1)\big) = (\mathtt{z}+1)!\}$
$\{[\mathtt{z} \mapsto \mathtt{z}+1]\big((\mathtt{y} \times \mathtt{z}) = \mathtt{z}!\big)\}$
$\mathtt{z} := \mathtt{z}+1;$
$\{(\mathtt{y} \times \mathtt{z}) = \mathtt{z}!\}$
$\{[\mathtt{y} \mapsto \mathtt{y}*\mathtt{z}](\mathtt{y} = \mathtt{z}!)\}$
$\mathtt{y} := \mathtt{y}*\mathtt{z}$
$\{\mathtt{y} = \mathtt{z}!\}$
$\}$
$\{(\mathtt{y} = \mathtt{z}!) \ \wedge \ \neg[\mathtt{z}!=\mathtt{x}]\}$
$\{(\mathtt{y} = \mathtt{z}!) \ \wedge \ \mathtt{z} = \mathtt{x}\}$
$\{\mathtt{y} = \mathtt{x}!\}$

Figure 3: Verification for Factorial

The key point here is that if we want to provide a precise specification for Factorial then we need to do something. In this case, any of the above solutions would be fine—as long as you clearly communicate your choice and what you mean by your notation. On a whim we will pick option (2), formalized as follows:

$$z! \quad = \quad \begin{cases} 1 & \text{when } z = 0 \\ z \times ((z-1)!) & \text{when } z > 0 \\ (-z)! & \text{when } z < 0 \end{cases}$$

Using this definition, we can formalize our desired postcondition as follows:

$$\mathtt{y} = \mathtt{x}! \quad \equiv \quad \exists n \ \big(\mathtt{y} \Downarrow n \ \wedge \ \mathtt{x} \Downarrow (n!)\big)$$

With this background material out of the way, in Figure 3 we verify Factorial given the precondition $\top$ and postcondition $\mathtt{y} = x!$.

$\{\top\}$
`while (true)`
{
 $\{\top \wedge [\texttt{true}]\}$
 `skip`
 $\{\top \wedge [\texttt{true}]\}$
 $\{\top\}$
}
$\{\top \wedge \neg[\texttt{true}]\}$
$\{\bot\}$

Figure 4: Verifying that an infinite loop can have postcondition $\bot$.

Exercise 2. ($\star$) The verification proof in Figure 3 is a standard proof from the literature used to teach Hoare logic, and is contained in a popular introductory textbook on formal methods. Unfortunately, it contains a flaw. What is it?

4 Total correctness

Sharp eyes might wonder about a problem with the While rule presented above. What happens if the program never terminates? In fact, using the While rule it is possible to get some strange results. Reexamine the verification for Factorial in Figure 3. Suppose we run this program from an initial context in which `x` is negative. In this case, the `while` loop will never terminate, since `z` starts at 0 and increments until it hits `x`—which it never will if `x` is negative.

We need to refine our understanding of what we mean by our Hoare triple $\{P\}\ c\ \{Q\}$. The postcondition Q need only hold **if the command c terminates**. If c never terminates, then we never reach a final context in which Q is supposed to hold, and so it does not actually matter what Q is. Program logics that do not guarantee termination are called logics of *partial correctness*. As a more extreme example of the cost/benefit of using a logic of partial correctness, consider the verification in Figure 4 that a simple infinite loop can be specified with postcondition $\bot$. Actually a postcondition of $\bot$ for a loop is not so confusing, since it means that the loop must never have terminated. The more reasonable-looking postcondition of the Factorial program is more problematic, since someone looking at the specification might not realize that the precondition of $\top$ is not enough to guarantee that Factorial will terminate.

The solution is to develop a Hoare logic of *total correctness*. We write[4]

$$[P]\ c\ [Q]$$

[4]We used a different notation to separate partial and total correctness in the lectures; either is fine as long as you are consistent.

$$\frac{}{[P]\ \texttt{skip}\ [P]}\ \textsf{TSkip}$$

$$\frac{}{[[x \mapsto e]\ P]\ x\ \texttt{:=}\ e\ [P]}\ \textsf{TAssignment}$$

$$\frac{P \to P' \qquad Q' \to Q \qquad [P']\ c\ [Q']}{[P]\ c\ [Q]}\ \textsf{TConsequence}$$

$$\frac{[P]\ c_1\ [Q] \qquad [Q]\ c_2\ [R]}{[P]\ c_1\texttt{;}\ c_2\ [R]}\ \textsf{TSequence}$$

$$\frac{[P \wedge [b]]\ c_1\ [Q] \qquad [P \wedge \neg[b]]\ c_2\ [Q]}{[P]\ \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}\ [Q]}\ \textsf{TIf}$$

Figure 5: Hoare axioms for total correctness (except for `while`)

to mean that given any starting context ρ_α that satisfies P, running the command c is guaranteed to terminate in some context ρ_ω that satisfies Q. Although we are not yet in a position to prove it, logics of total correctness are stronger than logics of partial correctness, in the sense that

$$\frac{[P]\ c\ [Q]}{\{P\}\ c\ \{Q\}}\ \textsf{Weaken}$$

A common but informal way to make the same point is that

$$\text{Partial Correctness} \quad + \quad \text{Termination} \quad = \quad \text{Total Correctness}$$

Except for the While rule, the axioms of a Hoare logic of total correctness are similar to the axioms of a Hoare logic of partial correctness; we give these rules in Figure 5. The difference is that we require that our subcommands terminate (*e.g.*, in TSequence we require $[P]\ c_1\ [Q]$ instead of $\{P\}\ c_1\ \{Q\}$), and produce a guarantee of termination (*e.g.*, $[P]\ c_1\texttt{;}\ c_2\ [R]$ instead of $\{P\}\ c_1\texttt{;}\ c_2\ \{R\}$).

When we get to developing a rule for the `while (`b`) {`c`}` command, things are a bit trickier. As we have observed, the previous rule does not guarantee termination, and merely knowing that the subcommand c terminates does not guarantee that the loop itself terminates. We will have to do something fancier.

A *termination measure* t is a function from contexts into the natural numbers. Suppose F is a formula in logic that has t as a free variable; then we can define an assertion $t@F$ that evaluates F after substituting the value of the termination measure (on the current context) for t as follows:

$$\rho \Vdash t@P \quad \equiv \quad [t \mapsto t(\rho)]P$$

This may be a bit confusing, so let us consider an example. Suppose our example termination measure t_a is the absolute value of program variable `a`:

$$t_a(\rho) \quad \equiv \quad |\rho(\texttt{a})|$$

Recall from earlier that in our example context ρ_a, variable `a` had value 3. Suppose our example formula is $F_a \equiv t_a < 5$. Then we can simplify as follows:

$$\rho_a \Vdash t_a @ F_a \quad = \quad [t_a \mapsto t_a(\rho_a)]F_a \quad = \quad [t_a \mapsto 3](t_a < 5) \quad = \quad 3 < 5$$

Since $3 < 5$, we conclude that on ρ_a, $t_a @ F_a$ holds.

Given this idea of termination measures, we are ready to define the Hoare rule of total correctness for the `while` command as follows:

$$\frac{\forall n \ \left(\ [P \wedge [b] \wedge t @ (t = n)] \ c \ [P \wedge t @ (t < n)] \ \right)}{[P] \ \texttt{while} \ (b) \ \{c\} \ [P \wedge \neg [b]]} \ \mathsf{TWhile}$$

This may look scary but it is not that bad. To conclude that the loop terminates, we require a somewhat stronger precondition above the bar. We need to prove that for all $n : \mathbb{N}$, if we start in a context that satisfies P, and in which b has evaluated to true, and in which the termination measure t is equal to n, then c will terminate in a context that satisfies P **and in which the termination measure t has strictly decreased**. The termination argument is roughly as follows: we start in some context ρ, in which the termination measure t is equal to some n. If the loop is finished (b evaluates to false), then we are done; if not, then our premise guarantees that executing c will strictly decrease the termination measure, and so we loop around and try again. Since the naturals are bounded below by 0, we know that we cannot strictly decrease the termination measure forever, so at some point we must terminate.

Using the TWhile rule can be tricky since you need to develop termination measures that decrease over each loop body; this is a major reason that we are willing to use logics of partial correctness, since at least they provide some guarantees without being quite as hard to use. In fact, there are many programs for which no-one has been able to figure out a reasonable termination measure.

Still, sometimes it is a very nice to be able to conclude that a program terminates; in Figure 6 we give an example of using the TWhile rule to prove the total correctness of the Parity function using the termination measure $t \equiv |\mathsf{y}|$. Notice how we introduce a fresh metavariable n in the body of the loop; the value of this metavariable is **constant** as we go around the loop so that we can verify that the termination measure has decreased at the end. We highlight the parts of the formulas that verify that the termination measure decreases.

5 Soundness of Hoare Logic

We have presented a series of commands, and two Hoare logics (one of partial and one of total correctness) that can be used to verify programs using those commands. Although we have tried to argue why the rules of those logics are reasonable, you may not be convinced that they are. This is a reasonable concern since it is quite easy to write down reasonable-looking Hoare rules that turn out to be unsound; for example, consider the following

$$\frac{[P \wedge [b]] \ c \ [P]}{[P] \ \texttt{while} \ (b) \ \{c\} \ [P \wedge \neg [b]]} \ \mathsf{BadTWhile}$$

$[\mathtt{x} \geq 0]$
$[[y \mapsto x](\mathtt{x} = \mathtt{y} \ \wedge \ \mathtt{y} \geq 0)]$
`y := x;`
$[\mathtt{x} = \mathtt{y} \ \wedge \ \mathtt{y} \geq 0]$
$[\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n) \ \wedge \ \mathtt{y} \geq 0)]$
`while (2<=y)`
{
$[(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0) \wedge [\mathtt{2<=y}] \wedge \boxed{t@(t = n)}]$
$[\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 2 \ \wedge \ \boxed{|\mathtt{y}| = n}]$
$[[\mathtt{y} \mapsto \mathtt{y-2}]\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0 \ \wedge \ \boxed{\mathtt{y} = n - 2}\Big)]$
`y := y-2`
$[\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0 \ \wedge \ \boxed{\mathtt{y} = n - 2}]$
$[\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0 \ \wedge \ \boxed{|\mathtt{y}| < n}]$
$[\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0 \ \wedge \ \boxed{t@(t < n)}]$
}
$[\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0\Big) \wedge \neg[\mathtt{2<=y}]]$
$[\Big(\exists n \ (\mathtt{x} \ = \ \mathtt{y} + (2 \times n)) \ \wedge \ \mathtt{y} \geq 0\Big) \wedge (\mathtt{y} < 2)]$
$[\big((\mathtt{y} \Downarrow 0) \wedge \mathsf{even}(\mathtt{x})\big) \vee \big((\mathtt{y} \Downarrow 1) \wedge \mathsf{odd}(\mathtt{x})\big)]$

Figure 6: Totally correct verification for Parity using $t \ = \ |\mathtt{y}|$

This is a seemingly-reasonable attempt to directly lift the While rule into the realm of total correctness. Of course, as discussed above, it fails: just because a loop body always terminates does not mean that the loop itself always terminates. Just consider the program `while (true) {skip}`!

In fact, as soon as one starts making a serious attempt to generalize our Hoare logics to more realistic languages (*e.g.*, with memory, input/output, concurrency), one needs to start writing down Hoare rules to reason about those new features. Since those new features are very complicated, it rapidly becomes very hard to be confident in their associated Hoare rules.

What is needed is a semantic model, and associated soundness proof, of Hoare logic. That is, we must provide a formal definition to the Hoare triple (the semantic model) and then prove each of our Hoare rules using our definition (the soundness proof). This is our remaining task today.

Operational semantics. A precise definition of the Hoare triple depends on a precise definition for how our language operates—in other words, on our language's *operational semantics.* To provide that operational semantics, we will define a three-place *step relation* of the form:

$$c \vdash \rho \rightsquigarrow \rho'$$

Here c is a command and ρ is a starting context; meanwhile, ρ' is an ending context—that is, the context that results from running c. Technically, the relation is defined as the least relation[5] satisfying the seven rules given below.

The eSkip rule formalizes the execution behavior of the `skip` command:

$$\frac{}{\texttt{skip} \vdash \rho \rightsquigarrow \rho} \ \mathsf{eSkip}$$

That is, given a starting context ρ, evaluating the `skip` command does nothing and returns the original context ρ. More interesting is the eAssign rule:

$$\frac{}{x \texttt{ := } e \vdash \rho \rightsquigarrow [x \mapsto \mathsf{neval}(\rho, e)]\rho} \ \mathsf{eAssign}$$

In words, executing the command x `:=` e transforms the starting context ρ into a new context ρ' that is equal to ρ everywhere except at x, where ρ' now contains the result of evaluating e. Hopefully this matches your intuition for x `:=` e.

We now turn to the commands that contain subcommands. The first is the eSequence rule, which specifies how two commands should be run in sequence:

$$\frac{c_1 \vdash \rho_1 \rightsquigarrow \rho_2 \qquad c_2 \vdash \rho_2 \rightsquigarrow \rho_3}{c_1\texttt{;}\ c_2 \vdash \rho_1 \rightsquigarrow \rho_3} \ \mathsf{eSequence}$$

To determine the result of c_1`;` c_2, you take the starting context ρ_1 and run c_1 until it results in the intermediate context ρ_2. Then, you take ρ_2 and run c_2 until it results in the final context ρ_3, which is the result of the entire sequence.

[5] *i.e.*, the relation is inductively defined.

We need two rules to define the behavior of `if (`b`) {`c_1`} else {`c_2`}`: one when b is true and the other when b is false. Here they are:

$$\frac{\mathsf{beval}(\rho, b) = \top \qquad c_1 \ \vdash \ \rho_1 \rightsquigarrow \rho_2}{\texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}} \ \vdash \ \rho_1 \rightsquigarrow \rho_2} \ \mathsf{eIf1}$$

$$\frac{\mathsf{beval}(\rho, b) = \bot \qquad c_2 \ \vdash \ \rho_1 \rightsquigarrow \rho_2}{\texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}} \ \vdash \ \rho_1 \rightsquigarrow \rho_2} \ \mathsf{eIf2}$$

The eIf1 rule say that if the boolean expression b evaluates to true ($\top$) in the starting context ρ_1, then the result of the entire conditional command (`if`) is found by running the first command c_1. The eIf2 rule covers the other case, when b evaluates to false ($\bot$); in this case we run the second command c_2.

Similarly, we need two rules to define the behavior of the `while` loop:

$$\frac{\mathsf{beval}(\rho, b) = \bot}{\texttt{while (}b\texttt{) \{}c\texttt{\}} \ \vdash \ \rho \rightsquigarrow \rho} \ \mathsf{eWhile1}$$

$$\frac{\mathsf{beval}(\rho, b) = \top \qquad c \ \vdash \ \rho_1 \rightsquigarrow \rho_2 \qquad \texttt{while (}b\texttt{) \{}c\texttt{\}} \ \vdash \ \rho_2 \rightsquigarrow \rho_3}{\texttt{while (}b\texttt{) \{}c\texttt{\}} \ \vdash \ \rho_1 \rightsquigarrow \rho_3} \ \mathsf{eWhile2}$$

The eWhile1 rule is straightforward enough: if the boolean expression b evaluates to false, then the loop will end and return the starting context ρ. The eWhile2 rule is a little more complicated, and has three premises. The first premise is quite simple: b must evaluate to true. The second premise says that we must then run the command c in the initial context ρ_1 to get to an intermediate context ρ_2; this is very similar to what we do in the eSequence, eIf1, and eIf2 rules: "recursing" on a subcommand. The third premise is the particularly interesting one: here we are **not** recursing on a subcommand, since we take the intermediate context ρ_2 and determine what final context ρ_3 the original loop `while (`b`) {`c`}` will produce. Since the step relation is inductively defined, we will only be able to recurse some finite number of times—eventually we must hit the "base case" of eWhile1. **The step relation will only be defined for loops that terminate.** Consider the infinite loop "`while (true) {skip}`"; given any starting context ρ, there is no context ρ' such that

$$\texttt{while (true) \{skip\}} \ \vdash \ \rho \rightsquigarrow \rho'$$

On the other hand, for terminating loops, the step relation computes the expected final context by "unrolling" the loop the required finite number of times.

All of this may seem a bit odd: essentially, we are formally saying that our programs have no semantics (that is, do not execute!) if they would run forever in the niëve intuition. Indeed, for more complicated programming languages, this style of step relation—called *big-step*—is inadequate. A big step operational semantics, like the one given above, computes the result of the entire program in one "big step": that is, the step relation takes a command, before-state, and after-state. This runs into lots of problems when we care about what the

program does along the way, such as input and output (to/from *e.g.*, keyboard, mouse, screen, printer, network). To reason about those kinds of languages, we need a more complicated *small-step* semantics, where each time we apply the step relation we compute the result of some simple command; in a small-step semantics, the step relation takes an input command and context, as we do here, but outputs **both** a next command and a next context[6]. For example, a small-step rule for the `skip` command might look as follows:

$$(\rho, \texttt{skip; } c) \rightsquigarrow (\rho, c)$$

If we ran the program for another step using the small-step semantics, it would start working on the command c. In a small-step semantics, infinite loops are easy to model and reason about since we can just keep on stepping forever.

Both partial and total Hoare triples only care about the final context if the program terminates. Therefore, it does not matter if we cannot model nonterminating programs with our semantics, and so for our simple language big step semantics are adequate (and simpler than small-step).

Exercise 3. Prove that the big-step relation defined by the seven rules eSkip—eWhile2 is deterministic. That is, if $c \vdash \rho_1 \rightsquigarrow \rho_2$ and $c \vdash \rho_1 \rightsquigarrow \rho_2'$, then $\rho_1' = \rho_2'$. Hint: do induction on the $c \vdash \rho \rightsquigarrow \rho'$ relation.

Exercise 4. Define a small-step operational semantics for our language. On terminating programs, this language should result in the same final context as the big-step semantics. You may find it convenient to require that all programs have a final `skip` command to signal when the small-step relation can stop.

Semantics of the Hoare triple. We turn to the final topic in our study of Hoare logic, which is providing a model for the Hoare triples themselves: that is, what does $\{P\}\ c\ \{Q\}$ mean? Actually, almost all of the heavy lifting has already been done. We begin by drawing on our connection to modal logic.

Our step relation, given a fixed command c, relates two worlds: the pre- and post-contexts of c. Thus, we define a relation S_c on contexts as follows:

$$\rho\ S_c\ \rho' \quad \equiv \quad c \vdash \rho \rightsquigarrow \rho'$$

Now that we have defined a relation on worlds, we can define the modal operators $\Box$ and $\Diamond$ (specialized to S and command c) in exactly the usual way:

$$\rho \Vdash \Box_c^S P \qquad \equiv \qquad \forall \rho' \ \big(\rho\, S_c\, \rho' \ \rightarrow \ \rho' \Vdash P\big)$$

$$\rho \Vdash \Diamond_c^S P \qquad \equiv \qquad \exists \rho' \ \big(\rho\, S_c\, \rho' \ \wedge \ \rho' \Vdash P\big)$$

What do these operators mean? If $\rho \Vdash \Box_c^S P$ holds, then P will hold in any final context reachable from ρ by running the command c—that is, if we run c, P

[6] Another option might be to define the big-step relation coinductively.

will hold. In contrast, if $\rho \Vdash \Diamond_c^S P$ holds, then **it is possible** to run c from the context ρ, and P will hold on the resulting context. Since our semantics do not allow us to run nonterminating programs, sometimes it is **not** possible to run c.

Now that we have expressed our modal operators, the definitions of the Hoare triples for partial and total correctness are easy to state:

$$\{P\}\ c\ \{Q\} \quad \equiv \quad \forall \rho\ \left(\rho \Vdash (P \to \Box_c^S Q)\right)$$

$$[P]\ c\ [Q] \quad \equiv \quad \forall \rho\ \left(\rho \Vdash (P \to \Diamond_c^S Q)\right)$$

Pleasingly, the definitions differ only by the modality they utilize. $\{P\}\ c\ \{Q\}$ means that given any context ρ on which P holds, Q will hold if we execute c. In contrast, $[P]\ c\ [Q]$ means that given any context ρ, if P holds on ρ, then we **can** execute c, and afterwards Q will hold. Since our semantics is only defined on terminating programs, a guarantee that c can run implies that c will terminate.

Exercise 5. Prove that total correctness is strictly stronger than partial correctness. That is, show that $[P]\ c\ [Q]$ implies $\{P\}\ c\ \{Q\}$. Note that you will need the fact that our step relation is deterministic (see Exercise 3).

Exercise 6. ($\star$) In the context of a nondeterministic semantics, our definition of Hoare triples for total correctness is not stronger than our definition for Hoare triples for partial correctness. Give a reasonable definition for Hoare triples of total correctness that is stronger in such a context.

The final task is to prove the various Hoare axioms given earlier (Skip, Assignment, Consequence, ..., While, TSkip, TAssignment, TConsequence, ..., TWhile) from the definitions. Most of these are quite simple.

Exercise 7. Prove all of the Hoare rules of partial correctness except While. You cannot just combine your proofs for exercises 5 and 8 since the **premises** of the rules of partial correctness allow nonterminating subcommands. You must prove the rules directly (but none of them should require a lots of work).

Exercise 8. Prove all of the Hoare rules of total correctness except TWhile. Again, none of them, individually, should require lots of work.

Exercise 9. ($\star$) Prove the While rule of partial correctness.

Exercise 10. ($\star$) Prove the TWhile rule of total correctness. Hint: do induction on the value of the termination measure.