Correctness ("Hoare") Triples, pt. 2

CS 536: Science of Programming, Fall 2019

9/19

A. Why

- To specify a program's correctness, we need to know its precondition and postcondition (what should be true before and after executing it).
- The semantics of a verified program combines its program semantics rule with the state-oriented semantics of its specification predicates.
- To connect correctness triples in sequence, we need to weaken and strengthen conditions.

B. Objectives

At the end of today you should know

- Programs may have many different annotations, and we might prefer one annotation over another (or not), depending on the context.
- Under the right conditions, correctness triples can be joined together.
- One general rule for reasoning about assignments goes "backwards" from the postcondition to the precondition.
- What weakening and strengthening are.

C. Examples of Partial and Total Correctness With Loops

- For the following examples, let $W \equiv$ **while** $k \neq 0$ **do** $k := k-1$ **od**.
  - **Example 1**: $\models_{tot} \{k \geq 0\}\ W\ \{k = 0\}$. If we start in a state where k is ≥ 0, then the loop is guaranteed to terminate in a state satisfying $k = 0$.
  - **Example 2**: $\models \{k = -1\}\ W\ \{k = 0\}$ but $\not\models_{tot} \{k = -1\}\ W\ \{k = 0\}$. The triple is partially correct but not totally correct because it diverges if $k = -1$. I.e., we have $\not\models_{tot} \{k = -1\}\ W\ \{T\}$. Also note that partial correctness would hold if we substitute any predicate for $k = 0$.
  - **Example 3**: $\models \{T\}\ W\ \{k = 0\}$ but $\not\models_{tot} \{T\}\ W\ \{k = 0\}$. The triple is partially correct but not totally correct because it diverges for at least one value of k.
- For the following examples, let $W' \equiv$ **while** $k > 0$ **do** $k := k-1$ **od**. (We're changing the loop test of W so that it terminates immediately when k is negative.)
  - **Example 4**: $\models_{tot} \{T\}\ W'\ \{k \leq 0\}$.
  - **Example 5**: $\models_{tot} \{k = c_0\}\ W'\ \{(c_0 \leq 0 \rightarrow k = c_0) \wedge (c_0 \geq 0 \rightarrow k = 0)\}$. This is Example 4 with the "strongest" (most precise) postcondition possible.

D. Typical Questions about Correctness Triples

Note: I'm assuming $\sigma \in \Sigma$, not $\Sigma_\perp$, σ is a state and not $\perp$.

- **If a triple is satisfied, is the precondition satisfied?**
  - Partial / total correctness in σ says something about how S behaves if $\sigma \vDash p$. But neither kind of correctness implies $\sigma \vDash p$.
- **If the precondition isn't satisfied, is the triple satisfied?**
  - If $\sigma \vDash \neg p$, then the triple is trivially satisfied (it meets the "no bug here" test), but we don't know anything about S actually behaves: We might get $\perp$, we might get a state $\tau \in \Sigma$ where $\tau \vDash q$, or we might get a state τ where $\tau \vDash \neg q$. So, both partial and total correctness hold.

E. Three Mostly Trivial Cases

There are three cases where partial correctness doesn't really tell us anything interesting. The first two cases are also uninteresting under total correctness, but the third one is actually informative under total correctness.

- **Case 1: *p* is a contradiction** (i.e., $\vDash \neg p$). Both kinds of correctness follow trivially because they are defined as "if $\sigma \vDash p$, then …," and since $\sigma \nvDash \neg p$, this always holds. Here's another way to think about this: If you assume false is true before running S, then you know false is true after running S, and false implies q.
  - **Example 6**: $\vDash$ `{F} x := 1 {2+2 = 5}` and $\vDash_{tot}$ `{F} x := 1 {2+2 = 5}`.
- **Case 2: *S* always causes an error**. In this case, partial correctness always holds but total correctness doesn't hold for states satisfying the precondition.
  - Partial correctness always holds because we never terminate in a state where q is false. Under total correctness, if S always causes an error, then the triple is satisfied only if the precondition is not satisfied.
  - **Example 7**: If $\Omega \equiv$ **`while true do skip od`**, then $\vDash \{p\}\ \Omega$ `{2+2 = 5}`. On the other hand, $\sigma \nvDash_{tot} \{p\}\ \Omega$ `{2+2 = 4}`. (Note: the triple is not simply invalid, it's never satisfied if $\sigma \vDash p$.)
- ***q* is a tautology** (i.e., $\vDash q$).
  - Partial correctness always holds: It holds if the program doesn't terminate; if it terminates in a state, that state always satisfies true. Total correctness is a bit different: It holds iff the program terminates.
  - **Example 8**: The triple $\{p\}\ S$ `{T}` is always valid under partial correctness. If the triple is valid under total correctness, it means that precondition p always ensures termination, which certainly is useful.
- Note in general, $\vDash_{tot} \{p\}\ S\ \{q\}$ iff $\vDash \{p\}\ S\ \{q\}$ and $\vDash_{tot} \{p\}\ S$ `{T}` (total correctness is partial correctness plus termination).

F. More Correctness Triple Examples

Same Code, Different Conditions

- The same piece of code can be annotated with conditions in different ways, and there's not always a "best" annotation. An annotation might be the most general one possible (we'll discuss this concept soon), but depending on the context, we might prefer a different annotation.

- Below, let $\mathtt{sum}(x, y) = x + (x+1) + (x+2) + \ldots + y$. (If $x > y$, let $\mathtt{sum}(x, y) = 0$.) In Examples 9 – 12, we have the same program annotated (with preconditions and postconditions) of various strengths (strength = generality).
- **Example 9**: {T} `i := 0; s := 0` {i = s = 0}.
  - This is the strongest (most precise) annotation for this program.
- **Example 10**: {T} `i := 0; s := 0` {i = s = 0 = sum(0, i)}.
  - This adds a summation relationship to `i` and `s` when they're both zero.
- **Example 11**: $\{n \geq 0\}$ `i := 0; s := 0` $\{0 = i \leq n \wedge s = 0 = \mathtt{sum}(0, i)\}$
  - This limits `i` to a range of values 0, , …, `n`. There's no way in the postcondition to know $n \geq 0$ unless we assume it in the precondition.
- **Example 12**: $\{n \geq 0\}$ `i := 0; s := 0` $\{0 \leq i \leq n \wedge s = \mathtt{sum}(0, i)\}$
  - The postcondition no longer includes `i = s = 0`, which might seem like a disadvantage but will turn out to be an advantage later.
- The next two examples relate to calculating the midpoint in binary search. Though the code is the same, whether the midpoint is strictly between the left and right endpoints depends on whether or not the endpoints are nonadjacent.
  - **Example 13**: $\{L < R \wedge L \neq R - 1\}$ $M := (L + R)/2$ $\{L < M < R\}$
  - **Example 14**: $\{L < R\}$ $M := (L + R)/2$ $\{L \leq M < R\}$
- DeMorgan's laws can apply when a test is a conjunction or disjunction.
- **Example 15**: Here we search downward for a nonnegative `x` where `f(x)` is ≤ `y`; we stop if `x` goes negative or we find an `x` with `f(x)` ≤ `y`.

```
{x ≥ 0}
while x ≥ 0 ∧ f(x) > y do x := x-1 od
{x < 0 ∨ f(x) ≤ y}
```

- **Example 16**: This is Example 15 rephrased as an array search; as long as we have a legal index `i` and `b[i]` isn't ≤ `y`, we move left. We stop if the index becomes illegal or we find an index with `b[i]` ≤ `y`.

```
{0 ≤ i}
while i ≥ 0 ∧ b[i] > y do i := i-1 od
{i < 0 ∨ b[i] ≤ y}
```

Joining Two Triples

- If the postcondition of one triple matches the precondition of another, we can join their programs and form a sequence: If $\{p_1\}\ S_1\ \{p_2\}$ and $\{p_2\}\ S_2\ \{p_3\}$ are satisfied in a state, so is $\{p_1\}\ S_1;\ S_2\ \{p_3\}$.
- **Example 17**: Note: `s = sum(0, k)` holds before and after the two assignments below, but not between them.

Combining	{s = sum(0, k)} s := s+k+1 {s = sum(0, k+1)}
and	{s = sum(0, k+1)} k := k+1 {s = sum(0, k)}
yields	{s = sum(0, k)} s := s+k+1; k := k+1 {s = sum(0, k)}

- **Example 18**: Alternatively, we can increment `k` first and then update `s`.

  Combining $\{s = \mathrm{sum}(0, k)\}$ `k := k+1` $\{s = \mathrm{sum}(0, k-1)\}$

  and $\{s = \mathrm{sum}(0, k-1)\}$ `s := s+k` $\{s = \mathrm{sum}(0, k)\}$

  yields $\{s = \mathrm{sum}(0, k)\}$ `k := k+1; s := s+k` $\{s = \mathrm{sum}(0, k)\}$

Reasoning About Assignments (Technique 1: "Backward")

- There are two general rules for reasoning about assignments.
- The first rule is a goal-directed one that works "backwards" — from the postcondition to the precondition.
- **Assignment Rule 1 ("Backward" assignment**: If $P(x)$ is a predicate function, then $\{P(e)\}\ v := e\ \{P(v)\}$. It turns out that $P(e)$ is the most general (the so-called "weakest") precondition that works with the assignment $v := e$ and postcondition $P(v)$. We'll study this in the next lecture.
- **Example 19**: $\{P(m/2)\}$ `m := m/2` $\{P(m)\}$
  - If $P(x) \equiv x > 0$, then this triple expands to $\{m/2 > 0\}$ `m := m/2` $\{m > 0\}$
- **Example 20**: $\{Q(k+1)\}$ `k := k+1` $\{Q(k)\}$
  - If $Q(x) \equiv s = \mathrm{sum}(0, x)$ then this triple expands to $\{s = \mathrm{sum}(0, k+1)\}$ `k := k+1` $\{s = \mathrm{sum}(0, k)\}$
- **Example 21**: $\{R(s+k+1)\}$ `s := s+k+1` $\{R(s)\}$
  - Say $R(x) \equiv x = \mathrm{sum}(0, k+1)$, then this triple expands to

    $\{s+k+1 = \mathrm{sum}(0, k+1)\}$ `s := s+k+1` $\{s = \mathrm{sum}(0, k+1)\}$
- In general, for $\{P(e)\}\ v := e\ \{P(v)\}$ to be valid, we need the following lemma:
- **Assignment Lemma**: For all σ, if $\sigma \vDash P(e)$ then $M(v := e, \sigma) = \sigma[v \mapsto \sigma(e)] \vDash P(v)$.
  - Intuitively, what this says is that if we want to know that v has property P after binding v to the value of e, we need to know that e has property P beforehand.

We won't go into the detailed proof of this lemma, but basically, you work recursively on the structures of $P(v)$ and $P(e)$ simultaneously. The important case is when we encounter an occurrence of v in $P(v)$ and the corresponding occurrence of e in $P(e)$. In $\sigma \vDash P(e)$, the value of e is $\sigma(e)$. In $\sigma[v \mapsto \sigma(e)] \vDash P(v)$, the value of v is also $\sigma(e)$. So intuitively, the truth value of $P(e)$ in σ should match the truth value of $P(v)$ in $\sigma[v \mapsto \sigma(e)]$.

G. Stronger and Weaker Predicates

- **Generalizing the Sequence Rule**: We've already seen that two triples $\{p\}\ S_1\ \{q\}$ and $\{q\}\ S_2\ \{r\}$ can be combined to form the sequence $\{p\}\ S_1\ ;\ S_2\ \{r\}$.
- Say we want to combine two triples that don't have a common middle condition, $\{p\}\ S_1\ \{q\}$ and $\{q'\}\ S_2\ \{r\}$.
  - We can do this iff $q \to q'$. If S_1 terminates in state τ and $\{p\}\ S_1\ \{q\}$ is valid, then $\tau \vDash q$. If $\vDash q \to q'$, then $\tau \vDash q'$, so if $\{q'\}\ S_2\ \{r\}$, then if running S_2 in τ terminates, it terminates in a state satisfying r.
  - This reasoning works both for partial and total correctness.
- Our earlier rule with $q \equiv q'$ is a special case of this more general rule (since q always implies itself).
- **Definition**: If $q \to r$ then q is **stronger than** r and r is **weaker than** q.

- I.e., the set of states that satisfy q is $\subseteq$ the set of states that satisfy r.
- (Technically, we should say "stronger than or equal to" and "weaker than or equal to," since we can have $q \leftrightarrow r$, but it's too much of a mouthful. Then "strictly stronger" means "stronger than or equal to and not equal to;" strictly weaker is similar.)
- **Note**: One notation for implication is $p \supset q$; it's important not to read that $\supset$ as a superset symbol, since the set of states for $p \subseteq$ the set of states for q.

- **Example 22**: `x = 0` is stronger than `x = 0` $\vee$ `x = 1`, which is stronger than `x` $\geq$ `0`.
- A predicate corresponds to the set of states that satisfy it, so we can use Venn diagrams with sets of states to display comparisons of predicate strength.
  - In Figure 1, $p \rightarrow q$ because the set of states for $p \subseteq$ set of states for q. We don't have $p \rightarrow r$, $q \rightarrow r$, $r \rightarrow p$, or $r \rightarrow q$, but we do have $p \rightarrow \neg r$ (all of p is outside r), $r \rightarrow \neg p$, and $\neg q \rightarrow \neg p$.

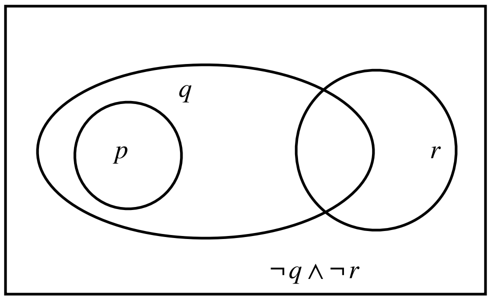

Figure 1: Predicates As Sets of States

H. Strengthening and Weakening Conditions

- This idea of freely going from stronger to weaker predicates applies to individual triples, not just sequences. Both of the following properties are valid for both partial and total correctness of triples (i.e., for $\vDash$ and $\vDash_{tot}$).
- **Preconditions can always be strengthened**: If $p_0 \rightarrow p_1$ and $\{p_1\}\ S\ \{q\}$, then $\{p_0\}\ S\ \{q\}$.
- **Postconditions can always be weakened**: If $q_0 \rightarrow q_1$ and $\{p\}\ S\ \{q_0\}$, then $\{p\}\ S\ \{q_1\}$.
- **Example 23**: If {`x` $\geq$ `0`} S {`y = 0`} is valid, then so are
  - {`x` $\geq$ `0`} S {`y = 0` $\vee$ `y = 1`} (Weak. postcondition)
  - {`x = 0`} S {`y = 0`} (Str. precondition)
  - {`x = 0`} S {`y = 0` $\vee$ `y = 1`} (Str. precond. and weak. postcond.)
- Note these rules still apply if $p_0 \leftrightarrow p_1$ or $q_0 \leftrightarrow q_1$.
- **Example 24**: Let $p_0 \equiv$ `s = sum(0, k)` and $p_1 \equiv$ `s+k+1 = sum(0, k+1)`
  Since $p_0 \leftrightarrow p_1$, we can "strengthen" $\{p_1\}\ S\ \{q\}$ to $\{p_0\}\ S\ \{q\}$ (where $q \equiv$ `s = sum(0, k)`).
- Just because we **can** strengthen preconditions and weaken postconditions doesn't mean we **should**. In the limit, strengthening preconditions and weakening postconditions gives us uninteresting correctness triples:
  - $\sigma \vDash$ {`F`} $S\ \{q\}$ and $\sigma \vDash_{tot}$ {`F`} $S\ \{q\}$ have the strongest possible preconditions

- $\sigma \models \{p\}\ S\ \{\mathtt{T}\}$ has the weakest possible postcondition
- $\sigma \models_{tot} \{p\}\ S\ \{\mathtt{T}\}$ has the weakest possible postcondition but it does tell us that S terminates when you start it in p.

- From the programmer's point of view, if $\{p\}\ S\ \{q\}$ has a bug, then strengthening p or weakening q can get rid of the bug without changing S.
  - **Example 25**: If $\{p\}\ S\ \{q\}$ causes an error if $\mathtt{x} = 0$, we can tell the user to use $\{p \wedge \mathtt{x} \neq 0\}\ S\ \{q\}$.
  - **Example 26**: If $\{p\}\ S\ \{q\}$ causes an error because S terminates with $\mathtt{y} = 1$ (and $\mathtt{y} = 1$ does not imply q), we can tell the user to use $\{p\}\ S\ \{q \vee \mathtt{y} = 1\}$.
- From the *user's* point of view, weaker preconditions and stronger postconditions make triples more useful:
  - Weaker preconditions make code more applicable by increasing the set of starting states. If $p \rightarrow q$ then going from $\{p\}\ S\ \{r\}$ to $\{q\}\ S\ \{r\}$ provides more information because it increases the set of states that we say S can start in and end with r. Basically, we combine $\{p\}\ S\ \{r\}$ and $\{\neg p \wedge q\}\ S\ \{r\}$ to get $\{q\}\ S\ \{r\}$. [9/18]
  - Stronger postconditions make code more specific by decreasing the set of ending states: If $q \rightarrow r$ then going from $\{p\}\ S\ \{r\}$ to $\{p\}\ S\ \{q\}$ provides more information because it decreases the set of states we say S will end in.

Correctness ("Hoare") Triples, pt. 2

CS 536: Science of Programming, Fall 2019

A. Why

- To specify a program's correctness, we need to know its precondition and postcondition (what should be true before and after executing it).
- The semantics of a verified program combines its program semantics rule with the state-oriented semantics of its specification predicates.
- To connect correctness triples in sequence, we need to weaken and strengthen conditions.

B. Objectives

At the end of today you should be able to

- Differentiate between different annotations for the same program.
- Determine whether two correctness triples can be joined and to give the result of joining.
- Reason "backwards" about assignment statements.
- Connect correctness triples in sequence by weakening and strengthening intermediate conditions

C. Questions

For Questions 1 – 5, use the triple {T} `y := x*x` {`y > x`}. There may be multiple correct answers.

1. This triple is not valid: Find one or more states in which it isn't satisfied.
2. Suggest a fix to the triple that involves changing only the precondition. That is, find a precondition that makes {`???`} `y := x*x` {`y > x`} valid.
3. Suggest a fix to the triple that involves changing only the program. That is, find a program that makes {T} `???` {`y > x`} valid.
4. Suggest a fix to the triple that involves changing only the postcondition. That is, find a postcondition that makes {T} `y := x*x` {`???`} valid.
5. If you didn't already, give the most precise postcondition possible for this program. (Just use your intuition. There's a formal framework for calculating this ("the strongest postcondition for the triple") which we'll look at in the future.)

For Questions 6 and 7, use the backward assignment rule discussed in the notes. There are multiple correct answers; any correct answer will do.

6. Find the most general precondition such that {`???`} `x := (x+1)/2` {$x \geq 0$} is valid.
7. Find the most general precondition such that {`???`} `y := 2*y` {`2*y < z`} is valid.

8. Let $S \equiv$ `x := x * x; y := y * y` and let $\sigma(x) = \alpha$ and $\sigma(y) = \beta$. Verify that $\sigma \vDash \{x > y > 0\}\ S\ \{x > y > 0\}$: Assume σ satisfies the precondition, calculate $M(S, \sigma)$, and verify that $M(S, \sigma) - \perp$ satisfies the postcondition.

[9/19: blank page removed]

Solution to Activity 9 (Hoare Triples, pt. 2)

1. We're looking for states that don't satisfy {`x*x` > `x`} `y := x*x` {`y` > `x`} — i.e., we need states where $x^2 \le x$; this includes `x` = `0` and `1`.

There exist many answers to Questions 2 – 5

2. {`x` > `1`} `y := x*x` {`y` > `x`}; some other answers `x` < `-1` and **`F`** (false). The most general precondition is $x \ne 0 \land x \ne 1$ [or anything ⇔ to it, like $x^2 > x$].
3. {**`T`**} **`if`** `x > 1` **`then`** `y := x*x` **`else`** `y := x+1` **`fi`** {`y` > `x`} is one possibility.
4. {**`T`**} `y := x*x` {$y \ge x$}; in general, any postcondition implied by $y = x^2$ works.
5. If we want to keep as much of the old postcondition as we can, one possibility is

   {**`T`**} `y := x*x` $\{(x \ne 0 \land x \ne 1 \rightarrow y \ge x) \land ((x = 0 \lor x = 1 \rightarrow y = x^2)\}$

   If just replace the postcondition, the most precise postcondition is $y = x^2$.

For Questions 6 and 7, these are the most general answers (again, up to ⇔)

6. $\{(x+1)/2 \ge 0\}$ `x := (x+1)/2` $\{x \ge 0\}$
7. $\{2*(2*y) < z\}$ `y := 2*y` $\{2*y < z\}$
8. We're given $S \equiv$ `x := x * x; y := y * y` and $\sigma(x) = \alpha$ and $\sigma(y) = \beta$.

   We can calculate $M(S, \sigma)$

   $= M($`x := x * x; y := y * y`$, \sigma)$

   $= M($`y := y * y`$, M($`x := x * x`$, \sigma))$

   $= M($`y := y * y`$, \sigma[x \mapsto \alpha^2])$)

   $= \{\tau\}$, where $\tau = \sigma[x \mapsto \alpha^2]\ [y \mapsto \beta^2]$

   If $\sigma \vDash x > y > 0$ then $\alpha > \beta > 0$, so $\alpha^2 > \beta^2 > 0$, and so $\tau \vDash x > y > 0$. Thus $\sigma \vDash \{x > y > 0\}\ S\ \{x > y > 0\}$: If $\sigma \vDash x > y > 0$ then $M(S, \sigma) - \perp \vDash x > y > 0$.

So if σ satisfies the precondition $x > y > 0$, then $M(S, \sigma)$ exists and satisfies the postcondition $x > y > 0$. Thus σ satisfies $\{x > y > 0\}\ S\ \{x > y > 0\}$.