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.

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.

C. Examples of Partial and Total Correctness With Loops

• For the following examples, let \( W \equiv \text{while } k \neq 0 \text{ do } k := k-1 \text{ od.} \)
  
  • Example 1: \( \models_{\text{tot}} \{ k \geq 0 \} W \{ k = 0 \}. \) If we start in a state where \( k \) is \( \geq 0 \), then the loop is guaranteed to terminate in a state satisfying \( k = 0 \).
  
  • Example 2: \( \not\models \{ k = -1 \} W \{ k = 0 \} \) but \( \not\models_{\text{tot}} \{ k = -1 \} W \{ k = 0 \}. \) The triple is partially correct but not totally correct because it diverges if \( k = -1 \).
  
  • Example 3: \( \not\models \{ T \} W \{ k = 0 \} \) but \( \not\models_{\text{tot}} \{ T \} W \{ k = 0 \}. \) The triple is partially correct but not totally correct because it diverges for at least one value of \( k \).

• For the following examples, let \( W' \equiv \text{while } k > 0 \text{ do } k := k-1 \text{ od.} \) (We’re changing the loop test of \( W \) so that it terminates immediately when \( k \) is negative.)
  
  • Example 4: \( \models_{\text{tot}} \{ T \} W' \{ k \leq 0 \}. \)
  
  • Example 5: \( \models_{\text{tot}} \{ k = c_0 \} W' \{ (c_0 \leq 0 \rightarrow k = c_0) \land (c_0 \geq 0 \rightarrow k = 0) \}. \) 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 \( \Sigma_{\bot} \). \( \sigma \) is a state and not \( \bot \).

• If a triple is satisfied, is the precondition satisfied?
  
  • Partial / total correctness in \( \sigma \) says something about how \( S \) behaves if \( \sigma \models p \). But neither kind of correctness implies \( \sigma \models p \).

• What if the precondition isn’t 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 \( \tau \) where \( \tau \vDash \neg q \). So, "Yes"

• If \( S \) (always) terminates and satisfies the postcondition, does the precondition matter?
  • Under both partial and total correctness, if \( M(S, \sigma) \vDash q \) for all \( \sigma \in \Sigma \) then the triple is satisfied whether \( \sigma \vDash p \) or \( \sigma \nvDash \neg p \). (Under total correctness, \( M(S, \sigma) \vDash q \) is exactly what we want; under partial correctness, we want \( M(S, \sigma) - \perp \vDash q \), which certainly holds if \( M(S, \sigma) \vDash q \) holds.) So, "No"

• What if \( S \) always causes an error (it always diverges or fails)?
  • Under partial correctness, if \( S \) always causes an error \( (M(S, \sigma) \subseteq \{ \perp_d, \perp_e \}) \), then the triple is satisfied whether \( \sigma \vDash p \) or \( \sigma \nvDash \neg p \). If \( S \) causes an error sometimes but terminates other times (both \( M(S, \sigma) \cap \{ \perp_d, \perp_e \} \) and \( M(S, \sigma) - \perp \) are nonempty), then we have to use the whole definition and say the triple is satisfied if \( \sigma \vDash \neg p \) or \( M(S, \sigma) - \perp \vDash q \). So "Yes" for partial correctness.
  • Under total correctness, if \( S \) can cause an error \( (M(S, \sigma) \cap \{ \perp_d, \perp_e \} \neq \emptyset) \), then the triple is satisfied iff \( \sigma \vDash \neg p \). So "Maybe" for total correctness.

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. For an alternative characterization, to think of this in terms of running \( S \), then if you assume false is true before running \( S \), then you know false is true after running \( S \), and false implies \( q \).
  • Example 6: \( \vDash \{ \text{F} \} \ x := 1 \{ 2+2 = 5 \} \) and \( \vDash_{\text{tot}} \{ \text{F} \} \ x := 1 \{ 2+2 = 5 \} \).

• Case 2: \( S \) always causes an error. Partial correctness always holds; total correctness never holds.
  • Partial correctness always holds because we never terminate in a state where \( q \) is false. I.e., given \( \sigma \vDash p \), we need \( \tau \vDash q \) for every \( \tau \in M(S, \sigma) - \perp \). Since \( S \) always causes an error, \( M(S, \sigma) - \perp \) is empty, and “if \( \tau \in \emptyset \) then \( \tau \vDash q \)” is vacuously true. So \( \sigma \vDash \{ p \} S \{ q \} \).
  • For total correctness, when \( \sigma \vDash p \), correctness never holds because we never terminate successfully. We need \( M(S, \sigma) \subseteq \Sigma \) and \( M(S, \sigma) \vDash q \). But \( M(S, \sigma) \subseteq \Sigma \) fails because it contains (only) error(s). So \( \sigma \nvDash_{\text{tot}} \{ p \} S \{ q \} \).
  • Example 7: If \( \Omega \equiv \text{while true do skip od} \), then \( \vDash \{ p \} \Omega \{ 2+2 = 5 \} \). On the other hand, \( \sigma \nvDash_{\text{tot}} \{ p \} \Omega \{ 2+2 = 4 \} \). (Note: the triple is not simply invalid, it’s never satisfied.)

• \( q \) is a tautology (i.e., \( \vDash q \)). Partial correctness always holds. Total correctness doesn't always hold, and when it does, it actually provides useful information: It says the program always terminates when started in \( \sigma \vDash p \).
  • For partial correctness, if \( S \) terminates in a state, then in that state, true holds. I.e., since every memory state satisfies \( q \), we get \( M(S, \sigma) - \perp \vDash q \).
  • Example 8: \( \vDash \{ p \} S \{ \top \} \).
• If $\sigma \models p$, then total correctness of \{p\} $S \{T\}$ says that the program successfully terminated (and here's the trivial part) in a state where true is satisfied. I.e., $\sigma \models_{\text{tot}} \{p\} S \{T\}$ iff $M(S, \sigma) \subseteq \Sigma$.

• This last case also illustrates the idea that total correctness is partial correctness plus guaranteed successful termination: If $\sigma \models p$, then $\sigma \models_{\text{tot}} \{p\} S \{q\}$ iff ($\sigma \models \{p\} S \{q\}$ and $\sigma \models_{\text{tot}} \{p\} S \{T\}$). For an example, consider Example 1 again: Let $W \equiv \text{while } k \neq 0 \text{ do } k := k-1 \text{ od}$, then total correctness holds because partial correctness and termination both hold: $\models_{\text{tot}} \{k \geq 0\} W \{k = 0\}$ because $\models \{k \geq 0\} W \{k = 0\}$ and $\models_{\text{tot}} \{k \geq 0\} W \{T\}$.

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 $\sum(x, y) = x + (x+1) + (x+2) + \ldots + y$. (If $x > y$, let $\sum(x, y) = 0$.) In Examples 19 – 22, 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: $\{n \geq 0\} i := 0; s := 0 \{0 = i \leq n \land s = 0 = \sum(0, i)\}$

- This limits $i$ to a range of values $0, \ldots, n$. There’s no way in the postcondition to know $n \geq 0$ unless we assume it in the precondition.

Example 12: $\{n \geq 0\} i := 0; s := 0 \{0 \leq i \leq n \land s = \sum(0, i)\}$

- 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: $\{L < R \land L \neq R - 1\} M := (L + R)/2 \{L < M < R\}$

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 $\leq y$; we stop if $x$ goes negative or we find an $x$ with $f(x) \leq y$.

$$\{x \geq 0\}$$

while $x \geq 0 \land f(x) > y$ do $x := x-1$ od

$$\{x < 0 \lor f(x) \leq y\}$$
• **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 \( \leq y \), we move left. We stop if the index becomes illegal or we find an index with \( b[i] \leq y \).

\[
\begin{align*}
0 & \leq i \\
\text{while } i & \geq 0 \land b[i] > y \text{ do } i := i - 1 \text{ od} \\
i & < 0 \lor b[i] \leq y
\end{align*}
\]

**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: \( s = \text{sum}(0, k) \) holds before and after the two assignments below, but not between them.

Combining \( \{ s = \text{sum}(0, k) \} s := s + k + 1 \{ s = \text{sum}(0, k + 1) \} \)

and \( \{ s = \text{sum}(0, k + 1) \} k := k + 1 \{ s = \text{sum}(0, k) \} \)

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

• **Example 18**: Alternatively, we can increment \( k \) first and then update \( s \).

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

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

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

**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 \( Q(x) \equiv s = \text{sum}(0, x) \) then this triple expands to \( \{ s = \text{sum}(0, k + 1) \} k := k + 1 \{ s = \text{sum}(0, k) \} \)

• **Example 21**: \( \{ R(s + k + 1) \} s := s + k + 1 \{ R(s) \} \)

  • Say \( R(x) \equiv x = \text{sum}(0, k + 1) \), then this triple expands to

    \( \{ s + k + 1 = \text{sum}(0, k + 1) \} s := s + k + 1 \{ s = \text{sum}(0, k + 1) \} \)

  • In general, for \( \{ P(e) \} v := e \{ P(v) \} \) to be valid, we need the following lemma:

• **Assignment Lemma**: For all \( \sigma \), if \( \sigma \models P(e) \) then \( M(v := e, \sigma) = \sigma[v \mapsto \sigma(e)] \models P(v) \).

  • 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 $\sigma \models P(e)$, the value of $e$ is $\sigma(e)$. In $\sigma[v \mapsto \sigma(e)] \models P(v)$, the value of $v$ is also $\sigma(e)$. So intuitively, the truth value of $P(e)$ in $\sigma$ should match the truth value of $P(v)$ in $\sigma[v \mapsto \sigma(e)]$. 
Correctness ("Hoare") Triples, pt. 2

CS 536: Science of Programming, Fall 2018

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.

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.

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 ≥ 0\} is valid.
7. Find the most general precondition such that \{???\} y := 2*y \{2*y < z\} is valid.

8. Let \(S \equiv x := x \times x; y := y \times y\) and let \(\sigma(x) = \alpha\) and \(\sigma(y) = \beta\). Verify that \(\sigma \models \{x > y > 0\} S \{x > y > 0\}\):
   Assume \(\sigma\) satisfies the precondition, calculate \(M(S, \sigma)\), and verify that \(M(S, \sigma) \perp \) satisfies the postcondition.
Solution to Activity 9 (Hoare Triples, pt. 2)

1. We're looking for states that don't satisfy \( x^2 > x \) — i.e., we need states where \( x^2 \leq x \); this includes \( x = 0 \) and \( 1 \).

There exist many answers to Questions 2 – 5.

2. \( \{ x > 1 \} \; y := x^2 \; \{ y > x \} \); some other answers \( x < -1 \) and \( \mathbf{F} \) (false). The most general precondition is \( x \neq 0 \land x \neq 1 \) [or anything \( \iff \) to it, like \( x^2 > x \)].

3. \( \{ \mathbf{T} \} \; \text{if } x > 1 \; \text{then } y := x^2 \; \text{else } y := x+1 \; \text{fi } \{ y > x \} \) is one possibility.

4. \( \{ \mathbf{T} \} \; y := x^2 \; \{ y \geq x \} \); 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

\[
\{ \mathbf{T} \} \; y := x^2 \; \{ (x \neq 0 \land x \neq 1 \rightarrow y \geq x) \land ((x = 0 \lor x = 1 \rightarrow y = x^2)) \}
\]

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 \( \iff \))

6. \( \{(x+1)/2 \geq 0 \} \; x := (x+1)/2 \; \{ x \geq 0 \} \)

7. \( \{ 2*(2*y) < z \} \; y := 2*y \; \{ 2*y < z \} \)

8. We're given \( S \equiv x := x \times x ; \; y := y \times y \) and \( \sigma(x) = \alpha \) and \( \sigma(y) = \beta \).

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

\[
= M(x := x \times x ; y := y \times y, \sigma)
= M(y := y \times y, M(x := x \times x, \sigma))
= M(y := y \times y, \sigma[x \mapsto \alpha^2])
= \{ \tau \}, \text{ where } \tau = \sigma[x \mapsto \alpha^2][y \mapsto \beta^2]
\]

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 \( M(S, \sigma) - \bot \vDash x > y > 0 \).

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