Correctness ("Hoare") Triples

Part 1: Definitions and Basic Properties

CS 536: Science of Programming, Fall 2022

2022-09-15 pp. 1, 6, 2022-09-20 p.7

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 joins a program's state-transformation semantics with the state-oriented semantics of the specification predicates.

B. Objectives

At the end of today you should know

- The syntax of correctness triples (a.k.a. Hoare triples).
- What it means for a correctness triples to be satisfied or to be valid.
- That a state in which a correctness triple is not satisfied is a state where the program has a bug.

C. Correctness Triples ("Hoare Triples")

- A ***correctness triple*** (a.k.a. "***Hoare triple***," after C.A.R. Hoare) is a program *S* plus its specification predicates *p* and *q*.
  - The ***precondition*** *p* describes what we're assuming is true about the state before the program begins.
  - The ***postcondition*** *q* describes what should be true about the state after the program terminates.
- ***Syntax of correctness triples***: *{p} S {q}* (Think of it as */* p */ S /* q */*)

  ⇒ *Note: The braces are not part of the precondition or postcondition* ⇐
- The precondition of *{p} S {q}* is *p*, not *{p}*. Similarly the postcondition is *q*, not *{q}*.
  - Saying "*{p}*" is like saying "In C, the test in '***if*** *(B) x++;*' is '***if*** *(B)*'" instead of just *B*. [2022-09-15]

D. Satisfaction and Validity of a Correctness Triple

- Informally, for a state to ***satisfy*** *{p} S {q}*, it must be that if we run *S* in a state that satisfies *p*, then after running *S*, we should be in a state that satisfies *q*.
  - There's more than one way to understand "after running *S*", and this will give us two notions of satisfaction.

- ***Important***: If we start in a state that doesn't satisfy p, we claim nothing about what happens when you run S.
  - In some sense, "the triple is satisfied in σ" means "the triple is not buggy in σ", which seems like a rather weak claim.
  - However, "the triple is not satisfied in σ" means "the triple has a bug in σ", which is a pretty strong statement.
- For example, say you're given the triple $\{x \geq 0\}\ S\ \{y^2 \leq x < (y+1)^2\}$.
  - The triple claims that running the program when x is nonnegative sets y to the integer square root of x.
  - If you run it when x is negative, all bets are off: S could run and terminate with y = some value, it could diverge, it could produce a runtime error. None of these behaviors are bugs because you ran S on a bad input.
- ***Validity*** for correctness triples is analogous to validity of a predicate: The triple must be satisfied in every (well-formed, proper) state.
  - Say you (as the user) have been told not to run S when $x < 0$ because S calculates $sqrt(x)$.
  - And say the triple is $\{x \geq 0\}\ y := sqrt(x)\ \{y^2 \leq x < (y+1)^2\}$.
  - You can't say this program has a bug when you start in a state with $x < 0$, even though the program fails, because you ran the program on bad input.
- ***Notation:*** Analogous to our notation for predicates, for triples
  - $\sigma \vDash \{p\}\ S\ \{q\}$ means σ satisfies the triple.
  - $\sigma \nvDash \{p\}\ S\ \{q\}$ means σ does not satisfy the triple.
  - $\vDash \{p\}\ S\ \{q\}$ means the triple is valid.
  - $\nvDash \{p\}\ S\ \{q\}$ means the triple is invalid: $\sigma \nvDash \{p\}\ S\ \{q\}$ for some σ.

E. Simple Informal Examples of Correctness

- Before going to the formal definitions of partial and total correctness, let's look at some simple examples, informally. (As usual, we'll assume the variables range over ℤ.)
- ***Example 1***: $\vDash \{x > 0\}\ x := x+1\ \{x > 0\}$. The triple is valid: It's satisfied for all states where $x > 0$.
- ***Example 2***:
  - $\{x = 1\} \nvDash \{x > 0\}\ x := x\text{-}1\ \{x > 0\}$: The triple is not satisfied (has a bug) when run with $x = 1$ because it terminates with $x = 0$, not > 0. Thus the triple is not valid: $\nvDash \{x > 0\}\ x := x\text{-}1\ \{x > 0\}$.
- There are a number of ways to fix the buggy program in Example 2:
  - ***Example 3***: Make the precondition "***stronger***" = "more restrictive": $\vDash \{x > 1\}\ x := x\text{-}1\ \{x > 0\}$.
  - ***Example 4***: Make the postcondition "***weaker***" = "less restrictive": $\vDash \{x > 0\}\ x := x\text{-}1\ \{x > -1\}$.
  - ***Example 5***: Change the program. One way is $\{x > 0\}$ ***if*** $x > 1$ ***then*** $x := x\text{-}1$ ***fi*** $\{x > 0\}$.
- Let's have some more complicated examples.

- ***Example 6***: $\models \{x \geq 0 \wedge (x = 2*k \vee x = 2*k+1)\}\ x := x / 2\ \{x = k \geq 0\}$.
  - If x is nonnegative, then the program halves it with truncation.
- ***Example 7***: $\models \{s = 0+1+2+ \dots +k\}\ s := s+k+1;\ k := k+1\ \{s = 0+1+2+ \dots +k\}$.
  - Note: strictly speaking, we need something like $s = sum(0, k)$ instead of $s = 1 + 2 + \dots + k$, which doesn't have the form of a predicate.
  - The triple says if $s = sum(0, k)$ when we start, then $s = sum(0, k)$ when we finish.
  - It's ok that s and k are changed by the program because $s = sum(0, k)$ is true in both places relative to the state at that point in time.
  - (Later, we'll use this program as part of a larger program, and we'll augment the conditions with information about how the ending values of k and s are larger than the starting values.)
- ***Example 8***: $\models \{s = sum(0, k)\}\ k := k+1;\ s := s+k\ \{s = sum(0, k)\}$
  - This has the same specification as Example 7 but the code is different: It increments k first and then update s by adding k (not $k+1$) to it.)
- ***Example 9***: [Note the invalidity] $\not\models \{s = sum(0, k)\}\ \ k := k+1;\ \ s := s+k+1\ \{s = sum(0, k)\}$
  - This is like Example 8 but the program doesn't meet its specification. To get validity, the postcondition should be $s = sum(0, k)+1$. (Or more likely, the code needs to be fixed.)

F. Connecting Starting and Ending Values of Variables

- There are times when we want the postcondition to be able to refer to values that the variables started with.
- Recall Examples 7 and 8: $\models \{s = sum(0, k)\}\ S\ \{s = sum(0, k)\}$ (where S is different in the two examples). Say we want the postcondition to include "k gets larger by 1" somehow. What we can do is create a new variable (call it k_0) whose job it is to refer to the value of k before S.
  - We'll make the precondition $k = k_0 \wedge s = sum(0, k)$ ("k has some value and s is the sum of 0 through k"). We'll make the postcondition $k = k_0+1 \wedge s = sum(0, k)$ ("k is one larger than its starting value and s is the sum of 0 through k (for this new value of k)".
- We actually did the same thing in Example 6: $\models \{x \geq 0 \wedge (x = 2*k \vee x = 2*k+1)\}\ x := x / 2\ \{x = k \geq 0\}$. The variable k helps describe the value of x before and after execution.
- One interesting feature of the variables k_0 and k is that they don't appear in the program, only the specifications.
- Where do variables appear in correctness triples?
- ***Definition:*** For a triple $\{\ p\ \}\ S\ \{\ q\ \}$,
  - A variable that appears in S is a ***program variable***. E.g., in $x := 1$, x is a program variable. We manipulate them to get work done.

- A variable that appears in p or q is a ***condition variable***. E.g., y in $\{ y > 0 \} \dots \{ \dots. \}$. We use condition variables to reason about our program. They may or may not also be program variables.[1]
  - E.g., in $\{ y > 0 \}\ y := y+1\ \{ y > 1 \}$, y is a program and a condition variable.
  - A ***logical variable*** is a condition variable that is not also a program variable. E.g., c in $\{ z \geq c \}\ z := z+1\ \{ z > c \}$. We use them to reason about our program but they don't appear in the program itself.[2]
  - A ***logical constant*** is a named constant logical variable. E.g., c in the previous example. Logical constants are great for keeping track of an old value of a variable.
- ***Example 10***: $\vDash \{x = x_0 \geq 0\}\ x := x / 2\ \{x_0 \geq 0 \wedge x = x_0 / 2\}$. If x is ≥ 0, then after the assignment $x := x/2$, the old value of x (we're calling it x_0) was ≥ 0 and x = its old value divided by 2. Here, x is a program and condition variable and x_0 is a logical constant.

G. Having a Set of States that Satisfy a Predicate

- Before looking at the definitions of program correctness, it will help if we extend the notion of a single state satisfying a predicate to having a set of states satisfying a predicate.
- ***Notation***: Recall that $\Sigma_\perp = \Sigma \cup \{\perp\}$, where Σ is the set of all (well-formed, proper) states.
  - Then, $\sigma \in \Sigma_\perp$ allows $\sigma = \perp$, but $\sigma \in \Sigma$ implies $\sigma \neq \perp$.
  - Similarly for a set of states Σ_0, if $\Sigma_0 \subseteq \Sigma_\perp$, then we may have $\perp \in \Sigma_0$.
  - On the other hand, if $\Sigma_0 \subseteq \Sigma$, then $\perp \notin \Sigma_0$.
- ***Notation***: $\Sigma_0 - \perp$ means $\Sigma_0 \cap \Sigma$, the subset of Σ_0 containing its non-$\perp$ members.
- ***Definition***: Let $\Sigma_0 \subseteq \Sigma_\perp$. We say Σ_0 ***satisfies*** p if every element of Σ_0 satisfies p.
  - In symbols, $\Sigma_0 \vDash p$ iff for all $\tau \in \Sigma_0$, $\tau \vDash p$. (Note $\varnothing \vDash p$, since there exists no $\tau \in \varnothing$ where $\tau \nvDash p$.)[3]
- Some consequences of the definition:
  - If $\perp \in \Sigma_0$, then $\Sigma_0 \nvDash p$ and $\Sigma_0 \nvDash \neg p$.
  - ($\Sigma_0 \vDash p$ and $\Sigma_0 \vDash \neg p$) iff $\Sigma_0 = \varnothing$.
    - Since $\perp \nvDash p$ (and $\nvDash \neg p$), we have $\perp \notin \Sigma_0$. If $\tau \neq \perp$ and $\tau \vDash p$ then $\tau \nvDash \neg p$, so $\tau \notin \Sigma_0$. So $\Sigma_0 = \varnothing$.
  - If $\perp \notin \Sigma_0$ and Σ_0 is a singleton set (it has size = 1), then $\Sigma_0 \vDash p$ iff $\Sigma_0 \vDash \neg p$ (and $\Sigma_0 \vDash \neg p$ iff $\Sigma_0 \nvDash p$).
    - Either $\tau \vDash p$ or $\tau \vDash \neg p$ but not both, so ($\tau \vDash p$ and $\tau \nvDash \neg p$) or ($\tau \nvDash p$ and $\tau \vDash \neg p$).
  - If $\Sigma_0 - \perp$ is not a singleton set then it is possible that $\Sigma_0 - \perp \nvDash$ both p and $\neg p$.
    - Say we have $\sigma_1, \sigma_2 \in \Sigma_0 - \perp$ where $\sigma_1 \vDash p$ and $\sigma_2 \vDash \neg p$. For $\Sigma_0 - \perp \vDash p$, we need all its members to satisfy p, but that's false, so $\Sigma_0 - \perp \nvDash p$. Similarly, $\Sigma_0 - \perp \nvDash \neg p$ because not all members of $\Sigma_0 - \perp$ satisfy $\neg p$.

---

[1] In distributed programming, "condition variable" has a related but different meaning.

[2] "Logical variable" here not the same as "boolean variable".

[3] If you run across an old set of these notes, you should know I changed how the notation works in F'20.

H. Total Correctness

- Normally, we want our programs to always terminate[4] in states satisfying their postcondition (assuming we start in a state satisfying the precondition). This property is called ***total correctness***.
- ***Definition***: The triple $\{p\}\ S\ \{q\}$ is ***totally correct in*** σ or σ satisfies the triple under ***total correctness*** iff it's the case that if σ satisfies p, then running S in σ always terminates in a state satisfying q.[5]
- In symbols, $\sigma \vDash_{tot} \{p\}\ S\ \{q\}$ iff $\sigma \neq \bot$ and (if $\sigma \vDash p$ then $\bot \notin M(S, \sigma)$ and $M(S, \sigma) \vDash q$).
  - The $\bot \notin M(S, \sigma)$ clause is redundant because $M(S, \sigma) \vDash q$ implies $\bot \notin M(S, \sigma)$.
- We specifically require $\sigma = \bot$ because $\bot \nvDash p$ and $M(S, \bot) = \{\bot\} \nvDash q$, so ($\sigma \vDash p$ implies $M(S, \sigma) \vDash q$) reduces to (false implies false), which is true.
- ***Definition***: The triple $\{p\}\ S\ \{q\}$ is ***totally correct*** (is ***valid*** under ***total correctness***) iff $\sigma \vDash_{tot} \{p\}\ S\ \{q\}$ for all $\sigma \in \Sigma$ (Recall Σ is the set of well-formed proper states.) Usually, we'll write $\vDash_{tot} \{p\}\ S\ \{q\}$.

I. Partial vs Total Correctness

- It turns out that reasoning about total correctness can be broken up into two steps: Determine "partial" correctness, where we ignore the possibility of divergence or runtime errors, and then show termination -- i.e., that those errors won't occur.
- ***Definition***: The triple $\{p\}\ S\ \{q\}$ is ***partially correct in*** σ or σ satisfies the triple under ***partial correctness*** iff $\sigma \neq \bot$ and if σ satisfies p, then whenever running S in σ terminates (without error), the final state satisfies q. Note if S diverges or causes a runtime error, we ignore those cases.
- In symbols, $\sigma \vDash \{p\}\ S\ \{q\}$ iff $\sigma \neq \bot$ and ($\sigma \vDash p$ implies (for every $\tau \in M(S, \sigma)$, if $\tau \in \Sigma$, then $\tau \vDash q$)).
- Equivalently, $\sigma \vDash \{p\}\ S\ \{q\}$ iff $\sigma \neq \bot$ and ($\sigma \vDash p$ implies $M(S, \sigma) - \bot \vDash q$).
- As with total correctness, we can't allow $\sigma = \bot$ for partial correctness because $\bot \nvDash p$, which would make ($\sigma \vDash p \Rightarrow \ldots$) true.
- ***Definition***: The triple $\{p\}\ S\ \{q\}$ is ***partially correct*** (i.e., is ***valid*** under/for ***partial correctness***) iff $\sigma \vDash \{p\}\ S\ \{q\}$ for all states σ. ***Notation***: We usually write $\vDash \{p\}\ S\ \{q\}$ but $\Sigma \vDash \{p\}\ S\ \{q\}$ is also ok.

J. More Phrasings of Total and Partial Correctness

- An equivalent way to understand partial and total correctness uses the property that if $\sigma \neq \bot$, then ($\sigma \vDash \neg p$ iff $\sigma \nvDash p$) and ($\sigma \vDash p$ iff $\sigma \nvDash \neg p$).

---

[4] "Terminate" will mean "terminate without error" (Final state $\in \Sigma - \bot$). "Terminate possibly with an error" means we end in $\Sigma_\bot$.

[5] The sense of "implies" or "if… then…" used here is not like → (which appears in predicates) or ⇒ (which is a relationship between predicates). It's "if…then" at a semantic level: If this triple is satisfied or if this set is nonempty, then … holds.

- For total correctness, if $\sigma \neq \bot$, then

  [2022-09-15] ~~*If σ ≠ ⊥, then*~~ $\sigma \models_{tot} \{p\}\ S\ \{q\}$
  iff $\sigma \models p$ implies $M(S, \sigma) \models q$
  iff $\sigma \models \neg p$ or $M(S, \sigma) \models q$
  iff $\sigma \models \neg p$ or $\tau \models q$ for every member $\tau \in M(S, \sigma)$
- Under total correctness, if S is deterministic, then there's only one $\tau \in M(S, \sigma)$, it's $\neq \bot$ and satisfies q. If S is nondeterministic, we can have multiple $\tau \in M(S, \sigma)$ and none of them can $= \bot$.
- For partial correctness, if $\sigma \neq \bot$, then

  $\sigma \models \{p\}\ S\ \{q\}$
  iff $\sigma \models p$ implies $M(S, \sigma) - \bot \models q$
  iff $\sigma \models \neg p$ or $M(S, \sigma) - \bot \models q$
  iff $\sigma \models \neg p$ or for every $\tau \in M(S, \sigma)$, either $\tau = \bot$ or $\tau \models q$.
- Under partial correctness, if S is deterministic, then there is only one τ in $M(S, \sigma)$ and either it $= \bot$ or satisfies q. If S is nondeterministic, we can have multiple $\tau \in M(S, \sigma)$ and all of them either $= \bot_d$ or $= \bot_e$ or satisfy q.

K. Unsatisfied Correctness Triples

- It's useful to figure out when a state ***doesn't satisfy*** a triple because not satisfying a triple tells you that there's some sort of bug in the program.

Unsatisfied Total Correctness

- For a state $\sigma \neq \bot$ to not satisfy $\{p\}\ S\ \{q\}$ under total correctness, it must satisfy p and running S in it can cause an error or one of its final states does not satisfy q.
  - We have $\sigma \models_{tot} \{p\}\ S\ \{q\}$ iff $\sigma \models \neg p$ or $M(S, \sigma) \models q$
  - So $\sigma \not\models_{tot} \{p\}\ S\ \{q\}$ iff $\sigma \models p$ and $M(S, \sigma) \not\models q$

    iff $\sigma \models p$ and ($\bot \in M(S, \sigma)$ or $\tau \not\models q$ for some $\tau \in M(S, \sigma)$ $\cancel{\models}$ [2022-09-15]).
  - (Recall if $\tau \neq \bot$ then $\tau \not\models q$ iff $\tau \models \neg q$.)
- If S is deterministic, then $\sigma \models p$ and $M(S, \sigma) = \{\tau\}$ where $\tau = \bot$ or $\tau \models \neg q$.
- If S is nondeterministic, then $\sigma \models p$ and ($\bot \in M(S, \sigma)$ or $\tau \models \neg q$ for some $\tau \in M(S, \sigma)$ $\cancel{\models}$ [2022-09-15]).
  - Note for $\sigma \not\models_{tot} \{p\}\ S\ \{q\}$, it's still possible to have $\tau \in M(S, \sigma)$ where $\tau \models q$ because we only need one $\tau \models \neg q$.

Unsatisfied Partial Correctness

- For a state $\sigma \neq \bot$ to not satisfy $\{p\}\ S\ \{q\}$ under partial correctness, it must satisfy p and running S in it always terminates in a state satisfying $\neg q$.
  - We have $\sigma \models \{p\}\ S\ \{q\}$ iff $\sigma \models \neg p$ or $M(S, \sigma) - \bot \models q$
  - So $\sigma \not\models \{p\}\ S\ \{q\}$ iff $\sigma \models p$ and $M(S, \sigma) - \bot \not\models q$
    iff $\sigma \models p$ and $\tau \models \neg q$ for some $\tau \neq \bot$ in $M(S, \sigma)$.

- For deterministic S, there's only one τ in $M(S, \sigma)$ and (it must be ≠ ⊥ and) satisfy $\neg q$.
- For nondeterministic S, we need one $\tau \in M(S, \sigma)$, ($\tau \neq \bot$ and) $\tau \vDash \neg q$.
    - The other $\tau \in M(S, \sigma)$ can be ⊥ or satisfy q.
    - I.e., at least one path $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ with $\tau \vDash \neg q$, but there can be paths $\langle S, \sigma\rangle \rightarrow^* \langle E, \bot\rangle$ or $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ with $\tau \vDash q$.

L. Three Extreme (Mostly Trivial) Cases

- There are three edge cases where partial correctness occurs for uninformative reasons.. First recall the definition of partial correctness: $\sigma \vDash \{p\}\ S\ \{q\}$ means (if $\sigma \vDash p$, then $M(S, \sigma) - \bot \vDash q$).
    - ***p is a contradiction*** (i.e., $\vDash \neg p$). Since $\sigma \vDash p$ never holds, $M(S, \sigma) - \bot \vDash q$ is irrelevant and partial correctness of $\{p\}\ S\ \{q\}$ always holds. So for example, $\{F\}\ S\ \{q\}$ is valid under partial correctness, for all S and q. (Even $\{F\}\ S\ \{F\}$ and $\{F\}\ S\ \{T\}$.)
    - ***S always doesn't terminate***[6]. If $M(S, \sigma) = \{\bot\}$ then $M(S, \sigma) - \bot = \varnothing \vDash q$, so we get partial correctness of $\{p\}\ S\ \{q\}$.
    - ***q is a tautology*** (i.e., $\vDash q$). Then for any σ, $M(S, \sigma) - \bot \vDash q$, so ($\sigma \vDash p$ implies $M(S, \sigma) - \bot \vDash q$) is true (so p is irrelevant) and we get partial correctness of $\{p\}\ S\ \{q\}$. So for example, $\{p\}\ S\ \{T\}$ is valid under partial correctness for all p and S. (Even $\{F\}\ S\ \{T\}$.)
- For total correctness, recall $\sigma \vDash_{tot} \{p\}\ S\ \{q\}$ means (if $\sigma \vDash p$, then $M(S, \sigma) \vDash q$). Note $\bot \notin M(S, \sigma)$ because $\bot \notin M(S, \sigma)$ implies $M(S, \sigma) \nvDash q$)
    - ***p is a contradiction***. The argument here is the same as for partial correctness, so for all S and q, we have $\vDash_{tot} \{F\}\ S\ \{q\}$.
    - ***S always doesn't terminate***. Since $M(S, \sigma) = \{\bot\}$, we know $M(S, \sigma) \nvDash q$. So total correctness of $\{p\}\ S\ \{q\}$ always fails. I.e., $\sigma \nvDash_{tot} \{F\}\ S\ \{q\}$ for all σ.
    - ***q is a tautology***. This case is actually useful. Since $M(S, \sigma) \vDash T$ implies $\bot \notin M(S, \sigma)$, satisfaction of $\sigma \vDash_{tot} \{p\}\ S\ \{T\}$ requires S ***to always terminate*** under σ. So validity of $\vDash_{tot} \{p\}\ S\ \{T\}$ happens when S always terminates when started in a state satisfying p.
- ***Lemma:*** $\sigma \vDash_{tot} \{p\}\ S\ \{q\}$ iff $\sigma \vDash \{p\}\ S\ \{q\}$ and $\sigma \vDash_{tot} \{p\}\ S\ \{T\}$.
    - This just says that total correctness is partial correctness plus termination.
    - Partial correctness says that $\langle S, \sigma\rangle \rightarrow^*$ to a final state that $\vDash q$ or is ⊥). Termination says every $\langle S, \sigma\rangle \rightarrow^*$ to a final state that satisfies true (and thus ≠ ⊥)). So we have total correctness: Every $\langle S, \sigma\rangle \rightarrow^*$ to a final state that $\vDash q$.

[6] Remember, just "terminate" implicitly includes "without error". "Not terminate" means "Diverges or gets a runtime error or whatever other flavor of ⊥ we have" [2022-09-20]

