Correctness ("Hoare") Triples

Part 2: Sequencing, Assignment, Strengthening, and Weakening

CS 536: Science of Programming, Fall 2022

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 strength is and 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*** $k := k\text{-}1$ ***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*** $k := k\text{-}1$ ***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***: $\models_{tot} \{k = c_0\}\ W'\ \{(c_0 \le 0 \rightarrow k = c_0) \wedge (c_0 \ge 0 \rightarrow k = 0)\}$. This is Example 4 with the "strongest" (most precise) postcondition possible. (In general, it's not always possible to find such a postcondition for loop, but it is here.)

D. 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 9 – 12, we have the same program annotated (with preconditions and postconditions) of various strengths (strength = generality).
- ***Example 9***: $\{T\}\ i := 0;\ s := 0\ \{i = 0 \wedge s = 0\}$.
  - This is the strongest (most precise) annotation for this program.
- ***Example 10***: $\{T\}\ i := 0;\ s := 0\ \{i = 0 \wedge s = 0 = sum(0, i)\}$.
  - This adds a summation relationship to *i* and *s* when they're both zero.
- ***Example 11***: $\{n \ge 0\}\ i := 0;\ s := 0\ \{0 = i \le n \wedge s = 0 = sum(0, i)\}$
  - This limits *i* to a range of values *0*, ..., *n*. We have to include $n \ge 0$ in the precondition if we want to claim $n \ge 0$ in the postcondition.
- ***Example 12***: $\{n \ge 0\}\ i := 0;\ s := 0\ \{0 \le i \le n \wedge s = sum(0, i)\}$
  - The postcondition no longer includes *i* and *s* being zero, so this postcondition is weaker (less precise) than the postcondition for Example 11. This 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***: $\{\ left < right \wedge left \ne right - 1\ \}\ mid := (\ left + right\ ) / 2\ \{\ left < mid < right\ \}$
  - ***Example 14***: $\{\ left < right\ \}\quad mid := (\ left + right\ ) / 2\ \{\ left \le mid < R\ \}$
- In Examples 13 and 14, the differences in the postcondition have an effect on how to detect that the value being searched for doesn't exist. In Example 13, it's $left = right - 1$; in Example 14, it's $left > right$.

Use of DeMorgan's Laws

- When a loop terminates, we know that the negation of the loop test holds, so DeMorgan's laws might be useful. Similarly, for a condition, we know that the negation of the test holds just before we execute the false branch.

- ***Example 15***: Here we search downward for $x \geq 0$ such that $f(x)$ is $\leq y$; we stop if we find such an x or if we run out of values to test.

  $\{x \geq 0\}$
  while $x \geq 0 \wedge f(x) > y$ ***do*** $x := x-1$ ***od***
  $\{x < 0 \vee f(x) \leq y\}$ *// Negation of loop test*

- ***Example 16***: This is Example 15 rephrased as an array search; we search to the left for an index k such that $b[k] \leq y$; we stop if we find one or run out of indexes to test.

  $\{k \geq 0\}$
  while $k \geq 0 \wedge b[k] > y$ ***do*** $k := k-1$ ***od***
  $\{k < 0 \vee b[k] \leq y\}$ *// Negation of loop test*

Joining Two Triples

- To make two statements a sequence, we have to compare the postcondition of the first statement and the precondition of the second. If they're the same, we can make the join.
  - I.e., if we have $\{p\}\ S_1\ \{q\}$ and $\{q\}\ S_2\ \{r\}$, then we can form $\{p\}\ S_1\ ;\ S_2\ \{r\}$ because when S_1 finishes executing, it will satisfy the precondition of S_2.
- ***Example 17***: We can join these two statements because the postcondition of the first statement matches the precondition of the second. (Note though $s = sum(0, k)$ holds before and after the two assignments, it doesn't hold between.)

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

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 $\{\ m/2 > 0\ \}\ m := m/2\ \{\ m > 0\ \}$.
  - If $P(x) \equiv x^2 > x^3$, then this triple expands to $\{\ (m/2)^2 > (m/2)^3\ \}\ m := m/2\ \{\ m^2 > m^3\ \}$.
- ***Example 20***: $\{\ Q(k+1)\ \}\ k := k+1\ \{\ Q(k)\ \}$
  - If $Q(x) \equiv s = sum(0, x)$ then this triple expands to $\{\ s = sum(0, k+1)\ \}\ k := k+1\ \{\ s = sum(0, k)\ \}$.

- ***Example 21***: *{ R(s+k+1) } s := s+k+1 { R(s) }*
  - If *R(x)* ≡ *x = sum(0, k+1)*, then this triple expands to
    { s+k+1 = sum(0, k+1) } s := s+k+1 { s = 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 *σ* ⊨ *P(e)* then *M(v := e, σ) = σ[v ↦ σ(e)]* ⊨ *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 *σ* ⊨ *P(e)*, the value of *e* is *σ(e)*. In *σ[v ↦ σ(e)]* ⊨ *P(v)*, the value of *v* is also *σ(e)*. So intuitively, the truth value of *P(e)* in *σ* should match the truth value of *P(v)* in *σ[v ↦ σ(e)]*.

E. 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* → *q′*. If S_1 terminates in state *τ* and *{p} S_1 {q}* is valid, then *τ* ⊨ *q*. If ⊨ *q* → *q′*, then *τ* ⊨ *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 discussion used a special case of the above. When *q* ≡ *q′*, we get that *{p} S_1 {q}* and *{q} S_2 {r}* can be joined.
- ***Definition***: If *p* → *q* then *p* is ***stronger than*** *q* and *q* is ***weaker than*** *p*.
  - I.e., the states that satisfy *p* also satisfy *q*.
    - (Technically, we should say "stronger than or equal to" and "weaker than or equal to," since we can have *p* ↔ *q*, but it's too much of a mouthful.
  - So *p* is ***strictly stronger*** than *q* and *r* is ***strictly weaker*** than *p* means *(p → q) ∧ ¬(q → p)*.
- ***Example 22***: *x = 0* is stronger than *x = 0 ∨ x = 1*, which is stronger than *x ≥ 0*.
- You can view a predicate as standing for the set of states that satisfy it. In that case, *p* → *q* means that the set of states for *p* is ⊆ the set of states for *q**.
- Venn diagrams with sets of states can help illustrate comparisons of predicate strength.

---

* One very old notation for implication is *p* ⊃ *q*; it's important not to read that ⊃ as a superset symbol, since *p* → *q* means that the set of states for *p* is ⊆ the set of states for *q*.

- In Figure 1, $p \rightarrow q$ because the set of states for p is $\subseteq$ the set of states for q. It's less obvious, but the contrapositive $\neg q \rightarrow \neg p$ also holds.
  - r has an intersection with q; the part inside q is $q \wedge r$; the part outside q is $\neg q \wedge r$.
  - p has no intersection with r, so neither $p \rightarrow r$ nor $r \rightarrow p$ hold, but all of p is outside r, so $p \rightarrow \neg r$ (and $r \rightarrow \neg p$).

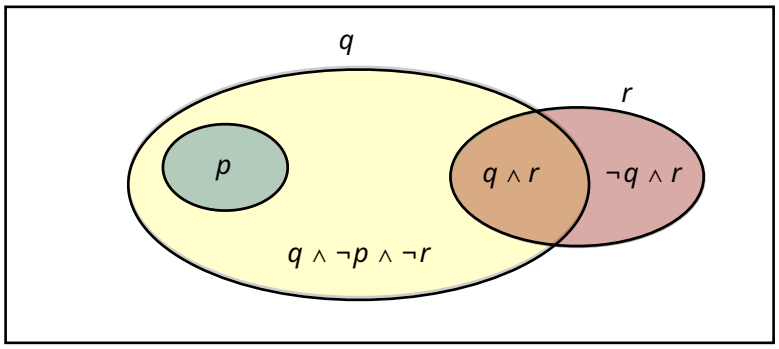

Figure 1: Predicates As Standing for Sets of States

F. Strengthening and Weakening Conditions

- The relationship between stronger and weaker predicates allows us to make "for free" certain changes to the conditions of triples. 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 $s \rightarrow b$ and $\{b\}\ S\ \{q\}$, then $\{s\}\ S\ \{q\}$ and
  - ***Postconditions can always be weakened***: If $s \rightarrow b$ and $\{p\}\ S\ \{s\}$, then $\{p\}\ S\ \{b\}$.
    - Read s and b as predicates with *smaller* and *bigger* sets of satisfying states.
    - The "always" above means it's sufficient to know $s \rightarrow b$ and the appropriate correctness triple — there's aren't any other restrictions.
- With p and q in Figure 1, $p \rightarrow q$, so p is stronger than q. Figure 1 illustrates how q were the precondition of a triple of it could be strengthened to p. On the other hand, if p were the postcondition of a triple, it could be weakened to q.
- ***Example 23***: If $\{x \geq 0\}\ S\ \{y = 0\}$ is valid, then so are
  - $\{x \geq 0\}\ S\ \{y = 0 \vee y = 1\}$ (Weakened postcondition)
  - $\{x = 0\}\ S\ \{y = 0\}$ (Strengthened precondition)
  - $\{x = 0\}\ S\ \{y = 0 \vee y = 1\}$ (Strengthened precondition and weakened postcondition)
- If the two conditions are equivalent (say $p \leftrightarrow p'$), we still call it "strengthening" and "weakening" since again, "stronger or equivalent" or "weaker or equivalent" is a bit too large of a mouthful.
- ***Example 24***: Since $s = sum(0, k) \Leftrightarrow s+k+1 = sum(0, k+1)$, we can "strengthen" the precondition of $\{\ s+k+1 = sum(0, k+1)\ \}\ \ k := k+1\ \{\ q\ \}$ and get $\{\ s = sum(0, k)\ \}\ \ k := k+1\ \{\ q\ \}$

Limitations of Strengthening and Weakening

- If $p \rightarrow q$ says that the sets of states satisfying p and q respectively are $\subseteq$, then the implications $q \rightarrow q_1$, $q_1 \rightarrow q_2$, $q_2 \rightarrow q_3$, ... form a sequence of weaker and weaker predicates. The sets of states gets larger and larger with the set Σ of all states as its limit. Since Σ satisfies T (true), true is the weakest possible predicate.
- Similarly, the implications ...., $p_3 \rightarrow p_2$, $p_2 \rightarrow p_1$, $p_1 \rightarrow p$ form a sequence of stronger and stronger predicates (when going to the left). Their sets of states get smaller and smaller with the empty set $\varnothing$ as its limit. As a set of states, this corresponds to F (false), so false is the strongest possible state.
- Having F be strongest and T be weakest may be counterintuitive. It may help if you think of the strength of a predicate being the amount of constraints it puts on the set of the states that satisfy it. No state satisfies the strongest predicate, F, and the set of all states satisfies the weakest predicate, T. Using set of states notation, $\{\ \} \vDash F$ and $\{\ \Sigma\ \} \vDash T$.

Can vs Should

- Just because we ***can*** strengthen preconditions and weaken postconditions doesn't mean we ***should***. Recall our edge cases for satisfying correctness triples:
  - $\sigma \vDash \{F\}\ S\ \{q\}$ and $\sigma \vDash_{\text{tot}} \{F\}\ S\ \{q\}$ have the strongest possible preconditions.
  - $\sigma \vDash \{p\}\ S\ \{T\}$ has the weakest possible postcondition.
  - $\sigma \vDash_{tot} \{p\}\ S\ \{T\}$ says that S terminates when you start it in p but it says nothing about what the state looks like when it terminates.
- 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 (Strengthening the precondition)***
    - If $\{p\}\ S\ \{q\}$ causes an error if $x = 0$, we can tell the user to use $\{p \wedge x \neq 0\}\ S\ \{q\}$.
  - ***Example 26 (Weakening the postcondition)***
    - If $\{p\}\ S\ \{q\}$ causes an error because S terminates satisfying predicate r (where r doesn't imply q) then we can tell the user to use $\{p\}\ S\ \{q \vee r\}$.

Weaker Preconditions and Stronger Postconditions

- From the user's point of view, weaker preconditions and stronger postconditions make triples more useful.
- ***Notation:*** Sometimes we'll abbreviate "*the set of states satisfying p*" to just "*p*".

Weaker Preconditions

- Weaker preconditions make code more applicable by increasing the set of starting states.
- Going back to Figure 1, say $\{p\}\ S\ \{u\}$ for some unshown predicate u. Certainly $\{q\}\ S\ \{u\}$ would give the programmer more flexibility in what states to start, but unlike strengthening a

precondition, weakening a precondition requires work: We must show that running starting with states in q but not p also can also be preconditions for $\{ \ldots \}\ S\ \{u\}$.

- Symbolically, if we know $\{p\}\ S\ \{u\}$ and can prove $\{q \wedge \neg p\}\ S\ \{u\}$, then combining these to get $\{\ p \vee (q \wedge \neg p)\ \}\ S\ \{u\}$, which simplifies to $\{q\}\ S\ \{u\}$.

Stronger Postconditions

- Stronger postconditions make code more specific by decreasing the set of ending states.
- Going back to Figure 1, say $\{w\}\ S\ \{q\}$ for some unshown predicate w. In contrast, $\{w\}\ S\ \{p\}$ is more precise about what the program does, but it requires us to show that there's no final state in $q \wedge \neg p$.
- Symbolically, if we know $\{w\}\ S\ \{q\}$ and $\{w\}\ S\ \{\ \neg(q \wedge \neg p)\ \}$ then we know $\{w\}\ S\ \{q \wedge \neg(q \wedge \neg p)\ \}$, which simplifies to $\{w\}\ S\ \{p\}$.

