Proof Rules and Proofs for Correctness Triples

Part 2: Conditional and Iterative Statements

CS 536: Science of Programming, Fall 2020

A. Why?

• We can't generally prove that correctness triples are valid using truth tables.
• We need inference rules for compound statements such as conditional and iterative.

B. Outcomes

At the end of this topic you should know

• The rules of inference for if-else statements.
• The rule of inference for while statements.
• The impracticality of the \( wp \) and \( sp \) for loops; the definition and use of loop invariants.

C. The If-Else (Conditional) Rule (and the If-Then Rule)

• The basic rule is

1. \( \{ p \land B \} S_1 \{ q_1 \} \)
2. \( \{ p \land \neg B \} S_2 \{ q_2 \} \)
3. \( \{ p \} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{ q_1 \lor q_2 \} \)

• Proof tree:

\[
\frac{\{ p \land B \} S_1 \{ q_1 \} \quad \{ p \land \neg B \} S_2 \{ q_2 \}}{\{ p \} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{ q_1 \lor q_2 \} \}
\]

conditional (or if-else) 1, 2

• (Notation: With our looser version of =, if \( q_1 = q_2 \), we can just write \( q_1 \) instead of \( q_1 \lor q_2 \).)

• The rule says that if you know that

  • Running the true branch \( S_1 \) in a state satisfying \( p \) and \( B \) establishes \( q_1 \)
  • And running the false branch \( S_2 \) in a state satisfying \( p \) and \( \neg B \) establishes \( q_2 \)
  • Then you know that running the if-else in a state satisfying \( p \) establishes \( q_1 \lor q_2 \).

Example 1:

1. \( \{ x \geq 0 \} y := x \{ y \geq 0 \} \)
2. \( \{ x < 0 \} y := -x \{ y \geq 0 \} \)
3. \( \{ T \} \text{if } x \geq 0 \text{ then } y := x \text{ else } y := -x \text{ fi } \{ y \geq 0 \} \)

• To get a full proof of the conclusion, we need to justify the assignments in lines 1 and 2:

1. \( \{ x \geq 0 \} y := x \{ y \geq 0 \} \)

2. \( \{ x < 0 \} y := -x \{ x < 0 \land y = -x \} \) assignment (forward)
3. \( x < 0 \land y = -x \rightarrow y \geq 0 \) predicate logic
4. \( \{ x < 0 \} y := -x \{ y \geq 0 \} \) postcondition weakening, 2, 3
5. \( \{ \top \} \text{if } x \geq 0 \text{ then } y := x \text{ else } fi \{ y \geq 0 \} \) conditional 1, 4

- There's an equivalent (but more general-looking) formulation of the conditional rule.
  1. \( \{ p_1 \} S_1 \{ q_1 \} \)
  2. \( q_1 \rightarrow q_1 \lor q_2 \) predicate logic
  3. \( \{ p_1 \} S_1 \{ q_1 \lor q_2 \} \) postcondition weakening 1, 2
  4. \( \{ p_2 \} S_2 \{ q_2 \} \)
  5. \( q_2 \rightarrow q_1 \lor q_2 \) predicate logic
  6. \( \{ p_2 \} S_2 \{ q_1 \lor q_2 \} \) postcondition weakening 4, 5
  7. \( p_0 \land B \rightarrow p_1 \) where \( p_0 = (p \land B \rightarrow p_1) \land (p \land \neg B \rightarrow p_2) \)
  8. \( p_0 \land \neg B \rightarrow p_2 \) predicate logic
  9. \( \{ p_0 \land B \} S_1 \{ q_1 \lor q_2 \} \) precondition strengthening 7, 3
  10. \( \{ p_0 \land \neg B \} S_2 \{ q_1 \lor q_2 \} \) precondition strengthening 8, 6
  11. \( \{ p_0 \} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{ q_1 \lor q_2 \} \) conditional 9, 10

**If-Then Statement Rule**

- We don't really need a separate rule for an if-then statement; we can treat it as an if-else skip: The \( \{ p \land \neg B \} S_2 \{ q_2 \} \) antecedent becomes \( \{ p \land \neg B \} \text{skip } \{ q_2 \} \), which can be proved if we know \( p \land \neg B \rightarrow q_2 \).
  1. \( \{ p \land B \} S_1 \{ q_1 \} \)
  2. \( p \land \neg B \rightarrow q_2 \)
  3. \( \{ q_2 \} \text{skip } \{ q_2 \} \) skip
  4. \( \{ p \land \neg B \} \text{skip } \{ q_2 \} \) precondition strengthening 2, 1
  5. \( \{ p \} \text{if } B \text{ then } S_1 \text{ fi } \{ q_1 \lor q_2 \} \) conditional (or if-else) 1, 4

**D. The While Loop (Iterative) Rule**

- The rule for while loops is different from the rules for the other statements because for those statements, the proof rule describes their wp/sp behavior. The rule for the while loop does not. (At least, not exactly.)
- We'll begin by looking at the rule and examples of it, then we'll look at the justification for a rule that uses a loop invariant to approximate the the wp/sp behavior of a loop.
A loop invariant \((p, \text{below})\) describes a correctness property that holds every time we encounter the loop test, either by falling into it for the first iteration, or jumping back up to it after we execute the loop body.

**Syntax:** We attach a keyword `inv` to the loop so that we can easily find the loop invariant when reading a program. The triple is written as `{inv p} while B do S od {p ∧ ¬B}`.

**While Loop Rule**

- The rule itself has one antecedent, which describes how the loop body behaves.
  1. `{p ∧ B} S {p}`
  2. `{inv p} while B do S od {p ∧ ¬B}`

- We ignore the `inv` keyword when using a loop triple so that \(p\) is the precondition for the triple; the triple behaves like `{p} while B do S od {p ∧ ¬B}`.

**Loop Initialization and Cleanup**

- We often use code to initialize loop variables before entering a loop; similarly, we can have code that cleans up any final details once the loop finishes. In general, we have `{p₀} initialization code; {inv p} while B do S od; {q} clean-up code`.
- The purpose of loop initialization is to establish the truth of the invariant before the first test of the loop: `{p₀} initialization code; {p}`. If there's no initialization code, we just need \(p₀ \rightarrow p\).
- The purpose of clean-up code is to take us from the loop postcondition to our desired final postcondition `{p ∧ ¬B} clean-up code {q}`. If there's no clean-up code, we just need \(p ∧ ¬B \rightarrow q\).

**E. Examples of Loops With An Invariant**

- Let \(W = \text{while } k < n \text{ do } k := k + 1; s := s + k \text{ od}\) and let \(p = 0 ≤ k ≤ n ∧ s = \text{sum}(0, k)\).
  - To use the loop rule, we need to show that its loop body takes us from \(p ∧ k < n\) back to \(p\).
  - I.e., we need to prove \({p ∧ k < n} k := k + 1; s := s + k {p}\). The correctness of the individual assignment statements will be proved using the assignment axiom. To combine them into a sequence, there are a couple of natural ways to do this, one using \(wp\) and one using \(sp\).

**Example 2: Verifying A Loop Body Using wp**

- Here is a proof that the loop body works, using \(wp\) and precondition strengthening.
  - (Recall \(p = 0 ≤ k ≤ n ∧ s = \text{sum}(0, k)\).)
  1. \(\{p[s+k/s]\} s := s + k {p}\) (backward) assignment
  2. \(\{p[s+k/s][k+1/k]\} k := k + 1 \{p[s+k/s]\}\) (backward) assignment
  3. \(\{p[s+k/s][k+1/k]\} k := k + 1; s := s + k {p}\) sequence 2, 1
  4. \(p ∧ k < n \rightarrow p[s+k/s][k+1/k]\) predicate logic
  5. \(\{p ∧ k < n\} k := k + 1; s := s + k {p}\) precondition str 4, 3

- (I put "backwards" in parentheses because the syntax of the triple only matches that version of the assignment rule. If you want to leave it out, that's fine.)
- To really believe that this is a proof, we have an obligation to believe the implication in line 4. First we need to expand the substitutions used:
• $p[s+k/s] = (0 \leq k \leq n \land s = \text{sum}(0, k)) \land s = \text{sum}(0, k)$

• $p[s+k/s][k+1/k] = (0 \leq k \leq n \land s+k = \text{sum}(0, k)) \land [k+1/k]
  = 0 \leq k+1 \leq n \land s+k+1 = \text{sum}(0, k+1)$

• Now it just requires some predicate logic to verify that $p \land k < n \rightarrow p[s+k/s][k+1/k]$.

• I.e., $(0 \leq k \leq n \land s = \text{sum}(0, k)) \land k < n \rightarrow (0 \leq k \leq n \land s = \text{sum}(0, k) \land k < n)$.

**Example 3: Correctness of a Loop Body Using $sp$**

• Here’s a proof of the same loop body triple $(p \land k < n) k := k+1; s := s+k \{ p \}$ using $sp$ and postcondition weakening. Syntactically, this proof is more complicated than the $wp$-using proof because we have to introduce names for the old values of $k$ and $s$.

• Again, recall that $p = 0 \leq k \leq n \land s = \text{sum}(0, k)$.

  1. $\{ p \land k < n \} k := k+1 \{ p_1 \}$ where $p_1 = (p \land k < n)[k_0/k] \land k = k_0+1$ assignment
  2. $\{ p_1 \} s := s+k \{ p_2 \}$ where $p_2 = p_1[s_0/s] \land s = s_0+k$ assignment
  3. $\{ p \land k < n \} k := k+1; s := s+k \{ p_2 \}$ sequence 1, 2
  4. $p_2 \rightarrow p$ predicate logic
  5. $\{ p \land k < n \} k := k+1; s := s+k \{ p \}$ post. weak. 4, 3

• Here are the expansions of $p_1$ and $p_2$:

  • $p_1 = (p \land k < n)[k_0/k] \land k = k_0+1$
    = $((0 \leq k \leq n \land s = \text{sum}(0, k)) \land k < n)[k_0/k] \land k = k_0+1$
    = $0 \leq k_0 \leq n \land s = \text{sum}(0, k_0) \land k_0 < n \land k = k_0+1$

  • $p_2 = p_1[s_0/s] \land s = s_0+k$
    = $(0 \leq k_0 \leq n \land s = \text{sum}(0, k_0) \land k_0 < n \land k = k_0+1)[s_0/s] \land s = s_0+k$
    = $0 \leq k_0 \leq n \land s_0 = \text{sum}(0, k_0) \land k_0 < n \land k = k_0+1 \land s = s_0+k$

• This time, our predicate logic obligation (in line 4) is $p_2 \rightarrow p$, which expands into

  $(0 \leq k_0 \leq n \land s_0 = \text{sum}(0, k_0) \land k_0 < n \land k = k_0+1 \land s = s_0+k) \rightarrow (0 \leq k \leq n \land s = \text{sum}(0, k))$

• The reasoning involved is still fairly simple, but definitely more complicated than in the $wp$ example.

**F. Modifying the Invariant Causes Changes to The Loop**

• The invariant, test, initialization code, and body of a loop are all interconnected: Changing one can change them all. For example, in Examples 2 and 3, we used $s = \text{sum}(0, k)$ in our invariant, so we wanted the loop to terminate with $k = n$.

• Relationships similar to $s = \text{sum}(0, k)$, like $s = \text{sum}(0, k+1)$ and $s = \text{sum}(0, k-1)$ make for perfectly good invariants, but using them introduces subtle changes to the initialization, body, and termination of the loop.

**Example 4**

• Instead of $s = \text{sum}(0, k)$, let’s use $s = \text{sum}(0, k+1)$. The loop terminates when $k+1 = n$, not $k = n$, so the range of $k$ changes. In addition the initialization of $s$ changes, and the loop body uses a different increment for $s$. 

© James Sasaki, 2020
\textbf{Example 5:}

- This time, instead of \( s = \text{sum}(0, k) \), let’s use \( s = \text{sum}(0, k-1) \). The loop terminates when \( k-1 = n \), so the range of \( k \) changes. Also, initialization of \( k \) changes, but the loop body is not changed.

\[
\{ n \geq 0 \} \quad \text{ // At minimum, } k-1 = 0 = n \\
\text{k := 1; s := 0;} \quad \text{ // } s = \text{sum}(0, k-1) = \text{sum}(0, 1-1) \\
\{ \text{inv } p'' \equiv 0 \leq k-1 \leq n \land s = \text{sum}(0, k-1) \} \quad \text{ // Terminate when } k-1 = n \\
\text{while } k \leq n \text{ do } \\
\quad \text{k := k+1; s := s+k-1} \quad \text{ // Need } s+k-1 \text{ before } k := k+1 \\
\text{od}
\]

\section*{G. The Theory Behind Loop Invariants}

\subsection*{Problems Calculating the sp or wp of a Loop}

- If \( W = \text{while } B \text{ do } S \text{ od} \) is a typical while loop, then what is the \( \text{sp}(p, W) \)? As it turns out, for some \( W \) and \( p \), we not may be able to write out \( \text{sp}(p, W) \) as a predicate. In that case, we can’t use \( \{ p \} W \{ \text{sp}(W, q) \} \) as the way to prove the partial correctness of a loop.

- To see how \( \text{sp}(p, W) \) might be impossible to write finitely, we can try to calculate it.
  - Let \( s_0 = p \) and for all \( k \geq 0 \), define \( s_{k+1} = \text{sp}(B \land s_k, S) \).
  - Semantically, say we run \( W \) starting in a state \( \tau_0 \), where \( \tau_0 \models s_0 \), then if \( \tau_0 \models B \), we run \( S \) and terminate in a state \( \tau_1 \in M(S, \tau_0) \), we know \( \tau_1 \models \text{sp}(B \land s_0, S) \), which is \( s_1 \).
  - More generally, if \( \tau_k \models B \land s_k \) then if we run \( S \) in \( \tau_k \) and terminate in \( \tau_{k+1} \), we know \( \tau_{k+1} \models \text{sp}(B \land s_k, S) \), which is \( \tau_{k+1} \).

- If the loop terminates, it terminates in the first \( \tau_k \) that satisfies \( \neg B \), and we get \( \text{sp}(p, W) \equiv s_{k+1} \).

- But there’s a problem. Unless we somehow know statically what \( s_{k+1} \) is (i.e., without running the program), we can’t statically write it out as a postcondition.
  - There are a couple of possibilities, but they don’t always apply.
    - One possibility is that somehow we know what \( k \) is. (This might happen, for example for a \textit{for} loop that runs a fixed number of times.)
    - Another possibility is that we might be able to show that at some point, \( s_{n+1} \Leftrightarrow s_n \), in which case, \( s_0 \Leftrightarrow s_{n+2} \Leftrightarrow \ldots \). (I.e., the sequence of \( s \) values has a finite limit.)

- But again, these don’t always happen.

- A similar problem comes up if we try to calculate \( \text{wp}(W, q) \) where we can come up with a sequence of partial results “the \textit{wp} if we run for 0 iterations”, “the \textit{wp} if we run for \( \leq 1 \) iteration”, “the \textit{wp} if

\textit{CS 536: Science of Programming} © James Sasaki, 2020
we run for \( \leq 2 \) iterations" and so on. There's no guarantee we can figure out which of these results is the correct one.

• And all this is was with leaving out the possibility that the loop diverges, in which case the \( sp \) and \( wp \) are both \( \equiv false \).

**Using Invariants to Approximate the \( wp \) and \( sp \) Of Loops**

• Since we can't calculate \( wp(S, q) \) or \( sp(p, W) \) exactly, the best we can do is to approximate it.
  • Instead of calculating the partial \( sp \) or \( wp \) results, we look for a predicate that encompasses all the results.
  • That's what a loop invariant does: It provides a general structure that describes all the possible states the loop might be when we hit the loop test.
    • At one extreme, this general structure is instantiated by a simple loop initialization.
    • At the other extreme, having the general structure and knowing that the loop test failed gives us enough information to conclude that our loop has done its work correctly.\(^1\)
    • For example, with the summation loop, the initialization code \( k := 0; s := 0 \) was an easy way to establish \( p = 0 \leq k \leq n \land s = \text{sum}(0, k) \), and combining \( p \) with the negation of the loop test implied that \( k = n \), and so \( s = \text{sum}(0, n) \), which is the final result we were looking for.
  • Semantically, if we go back to \( \tau_0, \tau_1, \tau_2, \ldots \), the sequence of states active when we encounter the while loop's test, the loop invariant is a property common to all these states, and it approximates various \( wp \) and \( sp \) for the loop:
    • \( p \) is an approximation of the loop's \( wp \): \( p \rightarrow wp(W, p \land \neg B) \).
    • \( p \land \neg B \) is an approximation of the loop's \( sp \): \( sp(p, W) \rightarrow p \land \neg B \).
    • \( p \) is an approximation of the \( wp \) of the loop body: \( p \land B \rightarrow wp(S, p) \).
    • \( p \) is an approximation of the \( sp \) of the loop body: \( sp(S, p \land B) \rightarrow p \).

---

\(^1\) By the way, for an arbitrary loop, there are an infinite number of loop invariants, so the trick isn't just finding one, it's finding one that's useful. We'll study techniques for developing loop invariants in a future class.