Proof Rules and Proofs for Correctness Triples

Part 1: Axioms, Sequencing, and Auxiliary Rules

CS 536: Science of Programming, Spring 2021

A. Why?
- We can’t generally prove that correctness triples are valid using truth tables.
- We need proof axioms for atomic statements (skip and assignment) and inference rules for compound statements like sequencing.
- In addition, we have inference rules that let us manipulate preconditions and postconditions.

B. Outcomes
At the end of this topic you should know
- The basic axioms for skip and assignment.
- The rules of inference for strengthening preconditions, weakening postconditions, composing statements into a sequence, and combining statements using if-else.

C. Truth vs Provability of Correctness Triples
- It’s time to start distinguishing between whether correctness triples are semantically true (valid) from whether they are provable: When we write a program, how can we convince ourselves that a triple is valid?
- In propositional logic, truth-validity is about truth tables, and proofs involve rules like associativity, commutativity, DeMorgan’s laws, etc.
- In predicate logic, truth-validity is about satisfaction of a predicate in a state, and one adds on rules to prove things about quantified predicates and about the kinds of values we’re manipulating.
  - We didn’t actually look at those rules specifically.
- In propositional logic, it’s often easier to deal with a truth table than to manipulate propositions using rules, but in predicate logic, proof rules are unavoidable because the truth table for a universal can be infinitely large.
  - (The truth table for \( \forall x \in S. P(x) \) has one row for each value of \( S \).)
- A proof system is a set of logical formulas determined by a set of axioms and rules of inference using a set of syntactic algorithms.
- One difference between validity and provability comes from predicate logic: Not everything that is true is provable.
• (This was proved by Kurt Gödel in the 1930s in his two Incompleteness Theorems.)
• Luckily, this problem doesn't come up in everyday programming (unless your idea of an everyday program involves writing programs that read programs and try to prove things about them).

• For correctness triples, the other difference comes from \textit{while} loops, basically because to describe the exact behavior of a loop may require an infinite number of cases.
  • (This is where undecidable functions come up in CS 530: Theory of Computing.)
  • Unfortunately, this problem really does come up in trying to prove that correctness triples are true.
  • Instead of proving the exact behavior of loops, we'll approximate their behavior using “loop invariants.” (Stay tuned.)

\textbf{D. Reasoning About Correctness Triples}

• So how do we reason about correctness triples?
  • First, we'll have \textbf{Axioms} that tell us that certain basic triples are true.
  • Second, we'll have \textbf{Rules of Inference} that tell us that if we can prove that certain triples are true, then some other triple will be true.

• In predicate logic we have axioms like “\( x + 0 = x \)” and rules of inference like \textit{Modus ponens}: If \( p \) and \( p \rightarrow q \), then \( q \) or \textit{Modus tollens}: If \( \neg q \) and \( p \rightarrow q \), then \( \neg p \).

• For a proof system for triples,
  • The formulas are correctness triples.
  • We'll have axioms for the \textit{skip} and assignment statements.
  • We'll have rules of inference for the sequence, conditional, and iterative statements.

• \textbf{Notation}: \( \vdash \{ p \} \ S \{ q \} \) means “We can prove that \( \{ p \} \ S \{ q \} \) is valid.”

• The \( \vdash \) symbol is a single turnstile pronounced “prove” or “can prove”.
• More generally, we might include predicates or correctness triples to the left of the turnstile. The meaning then is “If we assume (the items to the left of the turnstile) then we can prove that the right hand side triple is valid.”

\textbf{E. Proof System for Partial Correctness of Deterministic Programs}

• To get the proof rules, we'll follow the semantics of the different statements. That way we'll know the rules are \textit{sound} (if we can prove something, then it's valid). We won't try to deal with the opposite direction, \textit{completeness} (if something is valid, then we can prove it). We'll have one axiom or rule for each kind of statement and we'll have some auxiliary rules that don't depend on statements.
F. Proof Formats

Proof Trees

• In a proof tree, nodes are judgements: predicates for a predicate proof tree, correctness triples for a correctness triple proof tree. The set of edges between a parent and children are labeled by the logical rule that's being applied.

• The notation for a proof tree is not in computer science format. As examples, here is modus ponens on the left and multiplication by zero on the right:

\[
\begin{align*}
  p & \rightarrow q \\
  q & \text{modus ponens} \\
  x \times 0 &= x & \text{multiplication by zero}
\end{align*}
\]

• The rule name is attached to a line drawn between the parent and children. For a rule of inference, the antecedents (child judgements which are required to apply the rule) are written above the line; the consequent (the parent judgement that is being proved by the rule) is below. Leafs are judgements proved by axiom or assumption; they’re drawn with a labeled line above them but with no antecedents.

  • (This is the simplest kind of proof tree; there are more complicated kinds that include other features.)

• An advantage of proof trees is that you can read them top-down (“If we know \( p \) and we know \( p \rightarrow q \), then by modus ponens, we know \( q \”) or bottom-up (“To prove \( q \), by modus ponens it’s sufficient to prove \( p \) and \( p \rightarrow q \”).

• The main disadvantage of proof trees is that they’re difficult to draw. In a full proof, to use modus ponens, we would need a proof tree for \( p \) above it and a proof tree for \( p \rightarrow q \) above it, so they would be consequents of other rule applications, and their antecedents would be consequents in their own right, and so on. In a logic like that for correctness triples, nearly all the proof rules have multiple antecedents, so the lengthier the lines of reasoning become, the wider the tree becomes.

Hilbert-Style Proofs

• In a Hilbert-style proof, we have two columns. On the left, we have the judgements being proved; on the right we have the rule names being used to prove each judgement. Antecedents must above the consequent, and line numbers let us name the antecedents being used by a rule.

• In this format, modus ponens is

\[
\begin{align*}
  1. & \quad p \\
  2. & \quad p \rightarrow q \\
  3. & \quad q \quad \text{modus ponens 1, 2}
\end{align*}
\]

• The order of antecedents doesn’t matter; they just have to be above the consequent, so we could have written
1. \( p \rightarrow q \)
2. \( p \)
3. \( q \)

modus ponens 2, 1

- The name Hilbert-style proof comes from David Hilbert, one of the first people to investigate the structure of mathematical proofs.
- Below, we'll use Hilbert-style proofs because they are more convenient to write than proof trees and because it's the kind you're most likely to have seen before (in high-school geometry).

G. Skip Axiom

- Since the skip statement is a primitive statement, its correctness is proved by axiom.

\[
\text{Proof tree format:} \quad \text{Hilbert-style format:} \\
\hspace{1cm} \{p\} \text{ skip } \{p\} \quad \text{skip} \quad 1. \{p\} \text{ skip } \{p\} \quad \text{skip}
\]

H. Assignment Axioms

- The assignment statement is also primitive, so its also proved by axiom. Unlike skip, the assignment axiom comes in two versions.
- The \( wp \) version of assignment

\[
1. \{p[e/x]\} x := e \{p\} \quad \text{assignment (backward)}
\]

- The \( sp \) version of assignment

\[
1. \{p \land x = x_0\} x := e \{p[x_0/x] \land x = e[x_0/x]\} \quad \text{assignment (forward)}
\]

where \( x_0 \) is a fresh logical constant

- In addition, the \( x = x_0 \) clause is implied if omitted.
- Also, if \( x \) is itself fresh (doesn't appear in \( p \) or \( e \)), then the \( x = x_0 \) clause can definitely be dropped, and the rule simplifies to \( \{p\} x := e \{p \land x = e\} \) because we're simply introducing a new variable.

I. Sequence (= Composition) Rule

- The sequence rule allows us to take two statements and form a sequence from them.

\[
\text{Proof tree} \quad \text{Hilbert-style} \\
\hspace{1cm} \{p\} S_1 \{r\} \quad \{r\} S_2 \{q\} \quad \text{sequence} \quad 1. \{p\} S_1 \{r\} \\
\hspace{1cm} \{p\} S_1 ; S_2 \{q\} \quad \text{sequence 1, 2}
\]
Example 1:
- Below, lines 1 and 2 are proved by axiom, and line 3 is the use of the consequence rule.

1. \( \{T\} k := 0 \{ k = 0 \} \)  
   assignment
2. \( \{ k = 0 \} s := k \{ k = 0 \land s = 0 \} \)  
   assignment
3. \( \{T\} k := 0; s := k \{ k = 0 \land s = 0 \} \)  
   sequence 1, 2

J. Conjunction and Disjunction Rules
- The conjunction and disjunction rules are auxiliary rules that allow us to combine two proofs of the same triple. They're not connected to any particular statement, and their soundness relies on the semantics of \( \land \) and \( \lor \).

1. \( \{ p_1 \} S \{ q_1 \} \)
2. \( \{ p_2 \} S \{ q_2 \} \)
3. \( \{ p_1 \land p_2 \} S \{ q_1 \land q_2 \} \)  
   conjunction 1, 2

- Disjunction is similar:

1. \( \{ p_1 \} S \{ q_1 \} \)
2. \( \{ p_2 \} S \{ q_2 \} \)
3. \( \{ p_1 \lor p_2 \} S \{ q_1 \lor q_2 \} \)  
   disjunction 1, 2

- In the tree format, the conjunction rule is below. (Disjunction is similar.)

\[
\frac{\{ p_1 \} S \{ q_1 \} \quad \{ p_2 \} S \{ q_2 \}}{\{ p_1 \land p_2 \} S \{ q_1 \land q_2 \}} \quad \text{conjunction}
\]

K. Consequence Rule
- The consequence rule is also an auxiliary rule; it allows you replace a precondition and postcondition:

1. \( p_1 \rightarrow p_2 \)  
   predicate logic
2. \( \{ p_2 \} S \{ q_1 \} \)  
   ...
3. \( q_1 \rightarrow q_2 \)  
   predicate logic
4. \( \{ p_1 \} S \{ q_2 \} \)  
   consequence 1, 2, 3, 4

- Proof tree:

\[
\frac{p_1 \rightarrow p_2 \quad \{ p_2 \} S \{ q_1 \} \quad q_1 \rightarrow q_2}{\{ p_1 \} S \{ q_2 \}} \quad \text{consequence}
\]

- Note that two of the antecedents are said to be **predicate logic obligations** because they aren't correctness triples, they're predicate logic implications.
L. Strengthening and Weakening Rules

- Often (perhaps even usually) we only need the precondition or postcondition of a consequence rule. In that case we can use $T \rightarrow T$ as the other implication and either strengthen the precondition or weaken the postcondition.

**Strengthen Precondition**

1. $p_1 \rightarrow p_2$  
   predicate logic
2. $\{p_2\} S \{q\}$
3. $\{p_1\} S \{q\}$  
   strengthen precondition 1, 2

- Proof tree:

```
\[ p_1 \rightarrow p_2 \quad \{p_2\} S \{q\} \quad \{p_1\} S \{q\} \]
```

**Example 2:**

- (By adding a rule to justify line 2, we have a full proof here.)
1. $x \geq 0 \rightarrow x^2 \geq 0$  
   predicate logic
2. $\{x^2 \geq 0\} k := 0 \{x^2 \geq k\}$  
   assignment
3. $\{x \geq 0\} k := 0 \{x^2 \geq k\}$  
   precondition strengthen, 1, 2

- Proof tree:

```
x \geq 0 \rightarrow x^2 \geq 0 \quad \{x^2 \geq 0\} k := 0 \{x^2 \geq k\}
```

**Weaken Postcondition Rule**

- Symmetric to the precondition strengthening rule is the postcondition weakening rule. Like that rule, this one has a predicate logic obligation:
1. $\{p\} S \{q_1\}$
2. $q_1 \rightarrow q_2$  
   predicate logic
3. $\{p\} S \{q_2\}$  
   postcondition weakening 1, 2

- Proof tree:

```
\[ \{p\} S \{q_1\} \quad q_1 \rightarrow q_2 \quad \{p\} S \{q_2\} \]
```
Example 3:

This is a slightly different proof of the conclusion from Example 2.

1. \( \{ x \geq 0 \} \quad k := 0 \{ x \geq k \} \quad \text{assignment axiom} \\
2. \quad x \geq k \rightarrow x^2 \geq k \quad \text{predicate logic} \\
3. \{ x \geq 0 \} \quad k := 0 \{ x^2 \geq k \} \quad \text{postcond. weak. 1, 2}

- Proof tree:

\[
\begin{align*}
\{ x \geq 0 \} & \quad k := 0 \{ x \geq k \} \\
& \quad \text{assignment} \\
& \quad \begin{array}{c}
\{ x \geq 0 \} \quad k := 0 \{ x^2 \geq k \} \\
& \quad \text{predicate logic} \\
& \quad x \geq 0 \rightarrow x^2 \geq 0 \\
& \quad \text{postcondition weaken} \\
\end{array}
\end{align*}
\]

- In a correctness triple proof, there's often no unique proof of the conclusion, even ignoring how lines can be reordering. We can see this in Examples 2 and 3, which have slightly different predicate logic obligations, so they're certainly similar but not completely identical.

Example 4:

The conclusion of this proof appeared in Example 1.

1. \( \{ k = 0 \} \quad s := k \{ k = 0 \land s = k \} \quad \text{assignment (forward)} \\
2. \quad k = 0 \land s = k \rightarrow k = 0 \land s = 0 \quad \text{predicate logic} \\
3. \{ k = 0 \} \quad s := k \{ k = 0 \land s = 0 \} \quad \text{postcond. weak. 1, 2}

Example 5:

1. \( \{ 0 = 0 \land 0 = 0 \} \quad k := 0 \{ k = 0 \land k = 0 \} \quad \text{assignment (backwards)} \\
2. \quad T \rightarrow 0 = 0 \land 0 = 0 \quad \text{predicate logic} \\
3. \{ T \} \quad k := 0 \{ k = 0 \land k = 0 \} \quad \text{precondition strengthening 2, 1} \\
4. \{ k = 0 \land k = 0 \} \quad s := k \{ k = 0 \land s = 0 \} \quad \text{assignment (backwards)} \\
5. \{ T \} \quad k := 0; \quad s := k \{ k = 0 \land s = 0 \} \quad \text{sequence 3, 4}

Example 6:

- Here's another proof of the same conclusion that uses forward assignment instead of backwards assignment and postcondition weakening instead of precondition strengthening. It also uses the sequence rule earlier.

1. \( \{ T \} \quad k := 0 \{ T \land k = 0 \} \quad \text{assignment (forward)} \\
2. \{ T \land k = 0 \} \quad s := k \{ T \land k = 0 \land s = k \} \quad \text{assignment (forward)} \\
3. \{ T \} \quad k := 0; \quad s := k \{ T \land k = 0 \land s = k \} \quad \text{sequence 1, 2} \\
4. \quad T \land k = 0 \land s = k \rightarrow k = 0 \land s = 0 \quad \text{predicate logic} \\
5. \{ T \} \quad k := 0; \quad s := k \{ k = 0 \land s = 0 \} \quad \text{postcondition weakening 3, 4}
Technically, the “$T \land \_ k = 0 \ldots$” above needs to be there because it’s the “$p[v_0/v] \land \ldots$” part of the assignment rule, $\{p\} v := e \{p[v_0/v] \land v = e[v_0/v]\}$. But it’s annoying to write. But at this point, I think we’re familiar enough with syntactic equality that we can give ourselves a bit more freedom to abbreviate things.

**M. Looser Syntactic Equality**

- Recall that the purpose of syntactic equality is to give an easy way to guarantee semantic equality.
- Defining syntactic equality as ignoring redundant parentheses (including those from associative operators) was done because then determining syntactic equality is easy.
- Are there easy-to-detect logical transformations that can be added to parenthesis-removal that will let us weaken “syntactic” equality to “easily-provable semantic” equality?
  - E.g., in Example 6, postcondition weakening(-or-equal-to) tells us that $\{T\} k := 0 \{T \land k = 0\}$ and $\{T\} k := 0 \{k = 0\}$ are logically equivalent because $T \land k = 0 \iff k = 0$, which is very easy to prove.
- **Definition:** Two predicates are “loosely” syntactically equal if we can show that they are identical using the following transformations
  - *Ignoring redundant parentheses* (including those from associative operators).
  - *Identity:* $p \land T = p \lor F = p$.
  - *Domination:* $p \lor T = T$ and $p \land F = F$.
  - *Idempotence:* $p \lor p = p \land p = p$.
  - *Commutativity of $\land$ and $\lor$.
- Detecting the first two transformations is as easy as recognizing regular languages; detecting the second two can be done by combining the calculation of some normal form for predicates (for commutativity) with removal of duplicates (for idempotentcy).
- The final algorithm for detecting equality won’t be linear-time (as redundant-parenthesis removal is), but for small examples, at least, it should be usable.
- **Notation:** To make life more confusing¹, let’s continue to use $=\equiv$ but now mean this looser version of syntactic equality. We can say explicitly that some discussion needs the “strong” (original) version of $=\equiv$, if required.

¹ (Excuse me, I meant “more easy”.)
Proof Rules and Proofs for Correctness Triples

Part 1: Axioms, Sequencing, and Auxiliary Rules

CS 536: Science of Programming

A. Why

• We can’t generally prove that correctness triples are valid using truth tables.
• We need proof axioms for atomic statements (skip and assignment) and inference rules for compound statements like sequencing.
• In addition, we have inference rules that let us manipulate preconditions and postconditions.

B. Objectives

At the end of this practice activity you should

• Be able to match a statement and its conditions to its proof rule.

C. Problems

Use the vertical format to display rule instances. Below, ^ means exponentiation.

1. Consider the triples \{p_1\} x := x + x \{p_2\} and \{p_2\} k := k + 1 \{x = 2^k\} where \(p_1\) and \(p_2\) are unknown.
   a. Find values for \(p_1\) and \(p_2\) that make the triples provable. (Hint: Use \(wp\).)
   b. What do you get if you combine the triples using the sequence rule? Show the complete proof. (I.e., include the rules for the two assignments.)

2. Consider the incomplete triples \{p_2\} k := 0 \{p_1\} and \{p_1\} x := e \{x = 2^k\}, where \(p_1\), \(p_2\) and \(e\) are unknown.
   a. Find values for \(p_1\), \(p_2\) and \(e\) that make the two triples provable. Show the proofs. Hint: Calculate the \(wp\) of each assignment. Then stare at it to find values for \(e[0/k]\) and \(e\).
   b. If we combine the triples using the sequence rule, what results? Show the full proof.
Solution to Practice 14 (Proof Rules and Proofs, pt.1)

1. (Preconditions for \(x = 2^k\) postcondition)
   a. \(p_2 = wp(k := k+1, x = 2^k) = x = 2^{k+1}\). 
      \(p_1 = wp(x := x+x, p_2) = wp(x := x+x, x = 2^{k+1}) = x+x = 2^{k+1}\).
   b. The full proof is:
      1. \(\{x = 2^{k+1}\} k := k+1 \{x = 2^k\}\) assignment
      2. \(\{x+x = 2^{k+1}\} x := x+x \{x = 2^{k+1}\}\) assignment
      3. \(\{x+x = 2^{k+1}\} x := x+x; k := k+1 \{x = 2^k\}\) sequence 2, 1

2. (Initially establish \(x = 2^k\))
   a. Let \(p_1 = wp(x := e, x = 2^k) = x = 2^k[e/x] = e = 2^k\) and let \(p_2 = wp(k := 0, p_1) = (e = 2^k)[0/k] = (e[0/k] = 2^0)\). The simplest value for \(e\) is \(2^0\), since then \((e[0/k] = 2^0) = (2^0)[0/k] = 2^0 = 2^0\). But \(e = 1\) is more natural,
   b. The full proof is
      1. \(\{x = 2^0\} k := 0 \{x = 2^k\}\) assignment
      2. \(\{1 = 2^0\} x := 1 \{x = 2^0\}\) assignment
      3. \(\{1 = 2^0\} x := 1; k := 0 \{x = 2^k\}\) seq 2, 1

We can simplify \(1 = 2^0\) to \(7\) by using strengthening on the triple in line 1 or line 3.