Proof Rules and Proofs for Correctness Triples

Part 1: Axioms, Sequencing, and Auxiliary Rules

CS 536: Science of Programming, Spring 2023

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 mechanically 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 ***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.)

D. Reasoning About Correctness Triples

- So how do we reason about correctness triples, with proofs? We've been studying proofs formally, as textual things manipulated by textual operations (i. e. , proof rules).
  - First, we'll have ***Axioms*** that tell us that certain basic triples are true.
  - Second, we'll have ***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 ***Modus ponens***: If p and $p \rightarrow q$, then q or ***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 ***skip*** and assignment statements.
  - We'll have rules of inference for the sequence, conditional, and iterative statements.
- ***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". Often the $\vdash$ is left off as understood, but more generally, we can include items (predicates or correctness triples) to the left of the turnstile. This means "If we assume the items to the left then we can prove that the right hand side triple is valid. "

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 ***sound*** (if we can prove something, then it's valid). We won't try to deal with the opposite direction, ***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

- ***Definition: Judgements*** are the statements that proofs prove (or assume): In predicate logic, they're predicates; for us, they'll be correctness triples.
- There are various notations for writing out proofs. We'll show two of them (and you've already even seen one of them!).
- The first notation is a ***proof tree***. Here, each node is a judgement plus the name of a proof rule. Edges in the tree go from a parent to the children it relies on for use by the rule.
- For example, here are two rules you would expect in a predicate logic proof tree. On the left is ***modus ponens*** and on the right is multiplication by zero.

$$\frac{p \qquad p \rightarrow q}{q}\ \text{modus ponens} \qquad\qquad \frac{}{x * 0 = x}\ \text{multiplication by zero}$$

- 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 or bottom-up.
  - Top-down: If we know p and we know $p \rightarrow q$, then by modus ponens, we know q
  - 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, each antecedent that isn't proved by axiom is the root of its own proof subtree, and if a rule has more than one antecedent, then the tree becomes wider and wider.

Hilbert-Style Proofs

- In a Hilbert-style proof (the kind you've already seen), 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

  1. p
  2. $p \rightarrow q$
  3. q modus ponens 1, 2

- 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 people are generally more familiar with them from high-school geometry.
- For correctness triples, we'll have rules for each kind of statement (***skip***, assignment, etc.), plus a number of auxiliary rules that allow for manipulations like precondition strengthening, etc.

G. Skip Axiom

- The ***skip*** statement is a primitive statement (it contains no substatement), so its correctness is proved by axiom.

Proof tree format:

$$\frac{}{\{\, p \,\}\ \textbf{skip}\ \{\, p \,\}}\ \text{skip}$$

Hilbert-style format:

1. $\{\, p \,\}\ \textbf{skip}\ \{\, p \,\}$ skip

H. Assignment Axioms

- The assignment statement is also primitive, so it i also proved by axiom. Unlike skip, the assignment axiom comes in two versions.

- The *wp* version of assignment:

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

- The *sp* version of assignment:

  1. $\{\, p \wedge x = x_0 \,\}\ x := e\ \{\, p\,[\, x_0/x \,] \wedge x = e\,[\, x_0/x \,] \,\}$ assignment (forward)

     where x_0 is a fresh logical constant

- In addition, the $x = x_0$ clause is implied if omitted.
- 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 $\{ p \}\, x := e\, \{ p \wedge x = e \}$ because we're simply introducing a new variable.

I. Sequence (a.k.a. Composition) Rule

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

Proof tree

$$\frac{\{p\}\, S_1\, \{r\} \qquad \{r\}\, S_2\, \{q\}}{\{p\}\, S_1 ; S_2\, \{q\}}\ \text{sequence}$$

Hilbert-style

sequence

1. $\{p\}\, S_1\, \{r\}$
2. $\{r\}\, S_2\, \{q\}$
3. $\{p\}\, S_1 ; S_2\, \{q\}$ sequence 1, 2

Example 1:

- Below, lines 1 and 2 are proved by axiom, and line 3 is the use of the sequence rule.

1. $\{T\}\, k := 0\, \{k = 0\}$ assignment
2. $\{k = 0\}\, s := k\, \{k = 0 \wedge s = 0\}$ assignment
3. $\{T\}\, k := 0\, ;\ s := k\, \{k = 0 \wedge s = 0\}$ sequence 1, 2

J. Conjunction and Disjunction Rules

- ***Definition:*** An ***auxiliary*** proof rule is one that's not associated with proving a particular kind of statement.
- The conjunction and disjunction rules are auxiliary rules that allow us to combine two proofs of the same triple. Their soundness relies on the semantics of $\wedge$ and $\vee$, not on the semantics of statements, which is why they are auxiliary rules.

1. $\{p_1\}\, S\, \{q_1\}$
2. $\{p_2\}\, S\, \{q_2\}$
3. $\{p_1 \wedge p_2\}\, S\, \{q_1 \wedge q_2\}$ conjunction 1, 2

- Disjunction is similar:

1. $\{p_1\}\, S\, \{q_1\}$
2. $\{p_2\}\, S\, \{q_2\}$
3. $\{p_1 \vee p_2\}\, S\, \{q_1 \vee q_2\}$ disjunction 1, 2

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

$$\frac{\{p_1\}\, S\, \{q_1\} \qquad \{p_2\}\, S\, \{q_2\}}{\{p_1 \wedge p_2\}\, S\, \{q_1 \wedge q_2\}}\ \text{conjunction}$$

K. Strengthening and Weakening Rules

- The strengthening and weakening rules are also auxiliary rules. Unlike the other rules we've seen, they each include an antecedent that isn't a correctness triple; it's a predicate (an implication, specifically). Generically we call them ***predicate logic obligations***. We won't actually include predicate logic proofs of them in our correctness triple proofs, but the obligations do need to be true if our correctness triple proof is going to be true.

Strengthen Precondition

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

- Proof tree:

$$\frac{p_1 \to p_2 \qquad \{ p_2 \} S \{ q \}}{\{ p_1 \} S \{ q \}}\ \text{strengthen precondition}$$

Example 2:

- (By adding a rule to justify line 2, we have a full proof here.)

1.	$x \geq 0 \to x^2 > 0$	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:

$$\frac{\dfrac{}{x \geq 0 \to x^2 > 0}\ \text{predicate logic} \qquad \dfrac{}{\{ x^2 \geq 0 \}\ k := 0\ \{ x^2 \geq k \}}\ \text{assignment}}{\{ x \geq 0 \}\ k := 0\ \{ x^2 \geq k \}}\ \text{str. precondition}$$

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.	$q_1 \to q_2$	predicate logic
3.	$\{ p \} S \{ q_2 \}$	precond. weakening 1, 2

- Proof tree:

$$\frac{\{p\}\,S\,\{q_1\} \qquad q_1 \to q_2}{\{p\}\,S\,\{q_2\}}\ \text{postcondition weakening}$$

Example 3:

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

1.	$\{x \geq 0\}\ k := 0\ \{x \geq k\}$	assignment axiom
2.	$x \geq k \to x^2 > k$	predicate logic
3.	$\{x \geq 0\}\ k := 0\ \{x^2 > k\}$	postcond. weak. 1, 2

- Proof tree:

$$\frac{\dfrac{}{\{x^2 \geq 0\}\ k := 0\ \{x^2 \geq k\}}\ \text{assignment} \qquad \dfrac{}{x \geq 0 \to x^2 > 0}\ \text{predicate logic}}{\{x \geq 0\}\ k := 0\ \{x^2 \geq k\}}\ \text{postcondition weaken}$$

- 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.	$\{k = 0\}\ s := k\ \{k = 0 \wedge s = k\}$	assignment (forward)
2.	$k = 0 \wedge s = k \to k = 0 \wedge s = 0$	predicate logic
3.	$\{k = 0\}\ s := k\ \{k = 0 \wedge s = 0\}$	postcond. weak. 1, 2

Example 5:

1.	$\{0 = 0 \wedge 0 = 0\}\ k := 0\ \{k = 0 \wedge k = 0\}$	assignment (backwards)
2.	$T \to 0 = 0 \wedge 0 = 0$	predicate logic
3.	$\{T\}\ \ k := 0\ \{k = 0 \wedge k = 0\}$	precondition strength. 2, 1
4.	$\{k = 0 \wedge k = 0\}\ \ s := k\ \{k = 0 \wedge s = 0\}$	assignment (backwards)
5.	$\{T\}\ k := 0\ ;\ \ s := k\ \{k = 0 \wedge s = 0\}$	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.	$\{ T \} k := 0 \{ T \wedge k = 0 \}$	assignment (forward)
2.	$\{ T \wedge k = 0 \} s := k \{ T \wedge k = 0 \wedge s = k \}$	assignment (forward)
3.	$\{ T \} k := 0 ; s := k \{ T \wedge k = 0 \wedge s = k \}$	sequence 1, 2
4.	$T \wedge k = 0 \wedge s = k \rightarrow k = 0 \wedge s = 0$	predicate logic
5.	$\{ T \} k := 0 ; \ s := k \{ k = 0 \wedge s = 0 \}$	postcondition weakening 3, 4

- Technically, the "$T \wedge$ " part of "$T \wedge k = 0$ ..." above needs to be there because it's the "$p [v_0 / v] \wedge$ ..." part of the assignment rule, $\{ p \} v := e \{ p [v_0 / v] \wedge v = e [v_0 / v] \}$. 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.

L. Review - Looser Syntactic Equality

- (I know we talked about this in earlier classes but I thought it would be good to write it all down somewhere.)
- Syntactic equality, you'll recall, is used as an indicator of semantic equality. Checking for it is easy and fast but we give up complete accuracy: Syntactic equality implies semantic equality but not vice versa.
- We've been using a very simple notion of syntactic equality: Ignoring redundant parentheses. Though very fast (it takes linear time) and useful, it misses out on a number of properties that are reasonably easy to check for.
- We can trade in a bit of runtime for flexibility.

- ***Definition (Version 2 of syntactic equality***: Two predicates are syntactically equal if we can show that they are identical using the following transformations:
  - Ignore redundant parentheses (including those from associative operators).
  - Identity: $p \wedge T \equiv p \vee F \equiv p$.
  - Domination: $p \vee T \equiv T$ and $p \wedge F \equiv F$.
  - Idempotentcy: $p \vee p \equiv p \wedge p \equiv p$.
  - Commutativity of $\wedge$ and $\vee$.
- We can check for version 2 of $\equiv$ by converting two predicates into some sort of normal form first.
  - (Throw away identity and domination relations, sort the result to get around commutativity, and drop duplicates for idempotentcy.)

Example 6 revisited:

- Here's Example 6 written to use the looser notion of ≡.

1.	$\{ T \}\ k := 0\ \{ k = 0 \}$	assignment (forward)
2.	$\{ k = 0 \}\ s := k\ \{ k = 0 \wedge s = k \}$	assignment (forward)
3.	$\{ T \}\ k := 0\ ;\ s := k\ \{ k = 0 \wedge s = k \}$	sequence 1, 2
4.	$k = 0 \wedge s = k \rightarrow k = 0 \wedge s = 0$	predicate logic
5.	$\{ T \}\ k := 0\ ;\ \ s := k\ \{ k = 0 \wedge s = 0 \}$	postcondition weakening 3, 4