# *Proof Rules for Correctness Triples*

*CS 536: Science of Programming, Fall 2018*

10/9, 10/12 pp. 12,13

## *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 (sequencing, conditional, and iterative).

## *B. Outcomes*

At the end of this lecture 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`** and **`while`**.

## *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 $\forall \mathtt{x} \in S \,.\, P(\mathtt{x})$ 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?
  - 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 $p \to q$, then $q$ or *Modus tollens*: If $\neg q$ and $p \to q$, 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".
- 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."

### *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.

#### *Skip Axiom*

- **Skip axiom**: $\vdash \{p\}$ **`skip`** $\{p\}$ (for any predicate $p$).
- This rule is sound (i.e, $\vDash \{p\}$ **`skip`** $\{p\}$) because if $\sigma \vDash p$ as the precondition then $M(\textbf{skip}, \sigma) = \sigma \vDash p$ for the postcondition.

#### *Assignment Axioms*

- **Assignment axiom** (**backward**): $\vdash \{p[e/v]\}\ v := e\ \{p\}$
  - The soundness of this axiom follows from the validity of $\{p[e/v]\}\ v := e\ \{p\}$; i.e., for all $\sigma$, if $\sigma \vDash p[e/v]$, then $M(v := e, \sigma) = \sigma[v \mapsto \sigma(e)] \vDash p$.
- We have two additional "derived" rules that we can use like axioms. In both rules, a "fresh" symbol is one that doesn't appear free in the precondition or in the assignment.
- **Assignment** (**forward**): $\vdash \{p \wedge v = v_0\}\ v := e\ \{p[v_0/v] \wedge v = e[v_0/v]\}$, where $v_0$ is a fresh logical constant and the $v = v_0$ clause is implied if omitted.
- **Assignment** (**new variable**): $\vdash \{p\}\ v := e\ \{p \wedge v = e\}$, where $v$ is a fresh variable.

### *F. Rules of Inference*

- A rule of inference has the general form that if you can prove certain things (the **antecedents**) then you can prove some conclusion (the **consequent**). For rules that prove correctness triples to be valid, our consequent and (usually our) antecedents will be correctness triples.
- There are various ways to present proof rules and proofs. Common are proof trees and vertical proofs.
- **Proof trees**: Each rule is a node of a tree. The antecedents are written above the line; the consequent is below. In this format, *modus ponens* is

$$\frac{p \qquad p \to q}{q} \qquad \text{modus ponens}$$

  - In a full proof, each antecedent would be an axiom or the consequent of a proof rule. (So there would be lines above $p$ and $p \to q$ for the rules that let us conclude $p$ and $p \to q$).
  - Axioms and assumptions become childless nodes at the frontier of the tree.
  - A nice property of the tree format is that you can read a rule top-down or bottom-up:
    - "If we can prove $p$ and $p \to q$, then by modus ponens, we can prove $q$".
    - "If we want to prove $q$, then by modus ponens, it's sufficient to prove $p$ and $p \to q$".
- **Vertical proofs:** The antecedents and consequent are listed vertically, with the antecedents above the consequent. Line numbers let us name the antecedents. In this format, modus ponens is

| | | |
|---|---|---|
| 1. | $p$ | |
| 2. | $p \to q$ | |
| 3. | $q$ | modus ponens 1, 2 |

- Order is not important, so we could have written

| | | |
|---|---|---|
| 1. | $p \to q$ | |
| 2. | $p$ | |
| 3. | $q$ | modus ponens 2, 1 |

- In this format, a proof is a vertical list of **judgements** (the predicates or triples we're claiming are true) where each line follows from some axiom or assumption or from a rule of inference where the antecedents are somewhere above the consequent.
- These kinds of proofs are called "Hilbert-style" after David Hilbert, one of the first people to investigate the structure of mathematical proofs. We'll use this style of proof because it's the kind you're most likely to have seen before (in high-school geometry), and also because more convenient to write than the proof tree format.

### *Sequence Rule (a.k.a. Composition Rule)*

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

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

- The proof tree format is

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

- Just a reminder: You can read a rule top-down or bottom-up.
  - "If we can prove $\{p\}\ S_1\ \{r\}$ and $\{r\}\ S_2\ \{q\}$, then by the sequence rule, we can prove $\{p\}\ S_1\ \texttt{;}\ S_2\ \{q\}$".
  - "If we want to prove $\{p\}\ S_1\ \texttt{;}\ S_2\ \{q\}$, then by the sequence rule it's sufficient to prove $\{p\}\ S_1\ \{r\}$ and $\{r\}\ S_2\ \{q\}$ (for some $r$)".
- **Example 1:**

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

### *Conjunction and Disjunction Rules*

- The conjunction and disjunction rules are not connected to any particular statement; they are auxiliary rules that allow us to combine two proofs of the same triple. Their soundness relies on the semantics of $\wedge$ and $\vee$.

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.)

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

### *Consequence Rule*

- The consequence rule is also not connected to any particular statement; it's an auxiliary rule that 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

- Tree format:

$$\frac{p_1 \rightarrow p_2 \qquad \{p_2\}\ S\ \{q_1\} \qquad q_1 \rightarrow q_2}{\{p_1\}\ S\ \{q_2\}}\ \text{consequence}$$

 

- Note that two of the antecedents are predicate logic implications, not correctness triples. We call these **predicate logic obligations**.

### *Strengthen Precondition Rule*

- A special case of the Consequence rule concentrates only on strengthening the precondition, and it only introduces one predicate logic obligation:

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

(proof tree)

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

**Example 2**: (Below, we fill in a reason for believing line 2, so altogether, we have a complete proof.)

| | | |
|---|---|---|
| 1. | $\mathtt{x} \geq 0 \to \mathtt{x}^2 \geq 0$ | predicate logic |
| 2. | $\{\mathtt{x}^2 \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x}^2 \geq \mathtt{i}\}$ | assignment axiom |
| 3. | $\{\mathtt{x} \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x}^2 \geq \mathtt{i}\}$ | precondition strengthen, 1, 2 |

(proof tree):

$$\dfrac{\dfrac{}{\mathtt{x} \geq 0 \to \mathtt{x}^2 \geq 0}\ \text{predicate logic} \qquad \dfrac{}{\{\mathtt{x}^2 \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x}^2 \geq \mathtt{i}\}}\ \text{assignment}}{\{\mathtt{x} \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x}^2 \geq \mathtt{i}\}}\ \text{precondition strengthen}$$

### *Weaken Postcondition Rule*

- We can also use just the precondition weakening part of the Consequence rule. This rule also introduces one predicate logic obligation:

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

(proof tree)

$$\frac{\{p\}\ S\ \{q_1\} \qquad q_1 \to q_2}{\{p\}\ S\ \{q_2\}}$$

- **Example 3**: This is a slightly different proof of the conclusion from Example 2.

| | | |
|---|---|---|
| 1. | $\{\mathtt{x} \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x} \geq \mathtt{i}\}$ | assignment axiom |
| 2. | $\mathtt{x} \geq \mathtt{i} \to \mathtt{x}^2 \geq \mathtt{i}$ | predicate logic |
| 3. | $\{\mathtt{x} \geq 0\}\ \mathtt{i} := 0\ \{\mathtt{x}^2 \geq \mathtt{i}\}$ | postcond. weak. 1, 2 |

(proof tree):

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

 

- Often, there's not a unique proof of a triple, even if you don't worry about reordering the lines. 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. | $\{\texttt{i} = 0\}\ \texttt{s := i}\ \{\texttt{i} = 0 \wedge \texttt{s} = \texttt{i}\}$ | assignment (forward) |
| 2. | $\texttt{i} = 0 \wedge \texttt{s} = \texttt{i} \rightarrow \texttt{i} = 0 \wedge \texttt{s} = 0$ | predicate logic |
| 3. | $\{\texttt{i} = 0\}\ \texttt{s := i}\ \{\texttt{i} = 0 \wedge \texttt{s} = 0\}$ | postcond. weak. 1, 2 |

- **Example 5:**

| | | |
|---|---|---|
| 1. | $\{0 = 0 \wedge 0 = 0\}\ \texttt{i := 0}\ \{\texttt{i} = 0 \wedge \texttt{i} = 0\}$ | assignment (backwards) |
| 2. | $\texttt{T} \rightarrow 0 = 0 \wedge 0 = 0$ | predicate logic |
| 3. | $\{\texttt{T}\}\ \texttt{i := 0}\ \{\texttt{i} = 0 \wedge \texttt{i} = 0\}$ | precondition strengthening 2, 1 |
| 4. | $\{\texttt{i} = 0 \wedge \texttt{i} = 0\}\ \texttt{s := i}\ \{\texttt{i} = 0 \wedge \texttt{s} = 0\}$ | assignment (backwards) |
| 5. | $\{\texttt{T}\}\ \texttt{i := 0; s := i}\ \{\texttt{i} = 0 \wedge \texttt{s} = 0\}$ | 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. | $\{\texttt{T}\}\ \texttt{i := 0}\ \{\texttt{T} \wedge \texttt{i} = 0\}$ | assignment (forward) |
| 2. | $\{\texttt{T} \wedge \texttt{i} = 0\}\ \texttt{s := i}\ \{\texttt{T} \wedge \texttt{i} = 0 \wedge \texttt{s} = \texttt{i}\}$ | assignment (forward) |
| 3. | $\{\texttt{T}\}\ \texttt{i := 0; s := i}\ \{\texttt{T} \wedge \texttt{i} = 0 \wedge \texttt{s} = \texttt{i}\}$ | sequence 1, 2 |
| 4. | $\texttt{T} \wedge \texttt{i} = 0 \wedge \texttt{s} = \texttt{i} \rightarrow \texttt{i} = 0 \wedge \texttt{s} = 0$ | predicate logic |
| 5. | $\{\texttt{T}\}\ \texttt{i := 0; s := i}\ \{\texttt{i} = 0 \wedge \texttt{s} = 0\}$ | postcondition weakening 3, 4 |

  - Technically, the "$\texttt{T} \wedge$ " part of "$\texttt{T} \wedge \texttt{i} = 0$ …" 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.

### *A Looser Version of 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 $\{\texttt{T}\}\ \texttt{i := 0}\ \{\texttt{T} \wedge \texttt{i} = 0\}$ and $\{\texttt{T}\}\ \texttt{i := 0}\ \{\texttt{i} = 0\}$ are logically equivalent because $\texttt{T} \wedge \texttt{i} = 0 \Leftrightarrow \texttt{i} = 0$, 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*: $p \wedge \texttt{T} \equiv p \vee \texttt{F} \equiv p$.
  - *Domination:* $p \vee \texttt{T} \equiv \texttt{T}$ and $p \wedge \texttt{F} \equiv \texttt{F}$.

- *Idempotentcy:* $p \vee p \equiv p \wedge p \equiv p$.
- *Commutativity* of $\wedge$ and $\vee$.

- 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 confusing[1], 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.

### *Conditional/if-else Statement Rule*

- The basic rule is

  1. $\{p \wedge B\}\ S_1\ \{q_1\}$
  2. $\{p \wedge \neg B\}\ S_2\ \{q_2\}$
  3. $\{p\}$ **if** $B$ **then** $S_1$ **else** $S_2$ **fi** $\{q_1 \vee q_2\}$ conditional (or **if-else**) 1, 2

  or

  (proof tree)
  $$\frac{\{p \wedge B\}\ S_1\ \{q_1\} \qquad \{p \wedge \neg B\}\ S_2\ \{q_2\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q_1 \vee q_2\}}\ \text{conditional}$$

  - (Notation: With our looser version of $\equiv$, if $q_1 \equiv q_2$, we can just write $q_1$ instead of $q_1 \vee q_2$.)

- 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 $q_1 \vee q_2$.
- **Example 7**:

  1. $\{\mathtt{x} \geq 0\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} \geq 0\}$
  2. $\{\mathtt{x} < 0\}\ \mathtt{y} := -\mathtt{x}\ \{\mathtt{y} \geq 0\}$
  3. $\{\mathtt{T}\}$ **if** $\mathtt{x} \geq 0$ **then** $\mathtt{y} := \mathtt{x}$ **else** $\mathtt{y} := -\mathtt{x}$ **fi** $\{\mathtt{y} \geq 0\}$ conditional

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

  1. $\{\mathtt{x} \geq 0\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} \geq 0\}$ assignment (backward)
  2. $\{\mathtt{x} < 0\}\ \mathtt{y} := -\mathtt{x}\ \{\mathtt{x} < 0 \wedge \mathtt{y} = -\mathtt{x}\}$ assignment (forward)
  3. $\mathtt{x} < 0 \wedge \mathtt{y} = -\mathtt{x} \rightarrow \mathtt{y} \geq 0$ predicate logic
  4. $\{\mathtt{x} < 0\}\ \mathtt{y} := -\mathtt{x}\ \{\mathtt{y} \geq 0\}$ postcondition weakening, 2, 3
  5. $\{\mathtt{T}\}$ **if** $\mathtt{x} \geq 0$ **then** $\mathtt{y} := \mathtt{x}$ **else** $\mathtt{y} := -\mathtt{x}$ **fi** $\{\mathtt{y} \geq 0\}$ conditional 1, 4

[1] (Sorry, I meant "easier".)

 

- There's an equivalent (but more general-looking) formulation of the conditional rule.

| | | |
|---|---|---|
| 1. | $\{p_1\}\ S_1\ \{q_1\}$ | |
| 2. | $\{p_2\}\ S_2\ \{q_2\}$ | |
| 3. | $\{(p \wedge B \to p_1) \wedge (p \wedge \neg B \to p_2)\}$ **if** $B$ **then** $S_1$ **else** $S_2$ **fi** $\{q_1 \vee q_2\}$ | conditional |

- We can derive this second version of the conditional rule using the first version:

| | | |
|---|---|---|
| 1. | $\{p_1\}\ S_1\ \{q_1\}$ | |
| 2. | $q_1 \to q_1 \vee q_2$ | predicate logic |
| 3. | $\{p_1\}\ S_1\ \{q_1 \vee q_2\}$ | postcondition weakening 1, 2 |
| 4. | $\{p_2\}\ S_2\ \{q_2\}$ | |
| 5. | $q_2 \to q_1 \vee q_2$ | predicate logic |
| 6. | $\{p_2\}\ S_2\ \{q_1 \vee q_2\}$ | postcondition weakening 4, 5 |
| 7. | $s \wedge B \to p_1$ | predicate logic [10/9] |
| | where $s \equiv (p \wedge B \to p_1) \wedge (p \wedge \neg B \to p_2)$ | |
| 8. | $s \wedge \neg B \to p_2$ | predicate logic |
| 9. | $\{s \wedge B\}\ S_1\ \{q_1 \vee q_2\}$ | precondition strengthening 7, 3 |
| 10. | $\{s \wedge \neg B\}\ S_2\ \{q_1 \vee q_2\}$ | precondition strengthening 8, 6 |
| 11. | $\{s\}$ **if** $B$ **then** $S_1$ **else** $S_2$ **fi** $\{q_1 \vee q_2\}$ | conditional 9, 10 |

### *If-Then Statement Rule*

- [10/9] We don't really need a separate rule for an **if-then** statement; we can treat it as an **if-else skip**: The $\{p \wedge \neg B\}\ S_2\ \{q_2\}$ antecedent becomes $\{p \wedge \neg B\}$ **skip** $\{q_2\}$, which can be proved if we know $p \wedge \neg B \to q_2$:

| | | |
|---|---|---|
| 1. | $\{p \wedge B\}\ S_1\ \{q_1\}$ | |
| 2. | $p \wedge \neg B \to q_2$ | |
| 3. | $\{q_2\}$ **skip** $\{q_2\}$ | **skip** |
| 4. | $\{p \wedge \neg B\}$ **skip** $\{q_2\}$ | precondition strengthening 2, 1 |
| 5. | $\{p\}$ **if** $B$ **then** $S_1$ **fi** $\{q_1 \vee q_2\}$ | conditional (or **if-else**) 1, 4 |

### *While Loop Rule*

- The loop rule does not follow directly from the semantics of **while** loops because (as we'll see in a minute), there isn't any general way to write the *wp* or *sp* of a loop.
- Instead, the rule uses a predicate called the **loop invariant** ($p$, below), which is related to the *wp* and *sp*.

| | | |
|---|---|---|
| 1. | $\{p \wedge B\}\ S\ \{p\}$ | |
| 2. | $\{$**inv** $p\}$ **while** $B$ **do** $S$ **od** $\{p \wedge \neg B\}$ | loop (or **while**) |

- As a triple, the loop behaves like $\{p\}$ **while** $B$ **do** $S$ **od** $\{p \wedge \neg B\}$. In line 2, the keyword **inv** is used to indicate that $p$ is the loop invariant.

 

- **Example 7**: Let $p \equiv 0 \le \texttt{i} \le \texttt{n} \wedge \texttt{s} = \texttt{sum(0, i)}$ and $W \equiv$ **while** `i < n` **do** `i := i+1; s := s+i` **od**. To prove that $W$ works, we need to prove its loop body works: We need $\{p \wedge \texttt{i} < \texttt{n}\}$ `i := i+1; s := s+i` $\{p\}$. One way (there are other ways) to prove this is by using *wp* and precondition strengthening:

| | | |
|---|---|---|
| 1. | $\{p[\texttt{s+i/s}]\}$ `s := s+i` $\{p\}$ | assignment |
| 2. | $\{p[\texttt{s+i/s}][\texttt{i+1/i}]\}$ `i := i+1` $\{p[\texttt{s+i/s}]\}$ | assignment |
| 3. | $\{p[\texttt{s+i/s}][\texttt{i+1/i}]\}$ `i := i+1; s := s+i` $\{p\}$ | sequence 2, 1 |
| 4. | $p \wedge \texttt{i} < \texttt{n} \rightarrow p[\texttt{s+i/s}][\texttt{i+1/i}]$ | predicate logic |
| 5. | $\{p \wedge \texttt{i} < \texttt{n}\}$ `i := i+1; s := s+i` $\{p\}$ | precondition str 4, 3 |
| 6. | $\{$**inv** $p\}$ $W$ $\{p \wedge \texttt{i} \ge \texttt{n}\}$ | loop 5 |
| 7. | $p \wedge \texttt{i} \ge \texttt{n} \rightarrow \texttt{s} = \texttt{sum(0, n)}$ | predicate logic |
| 8. | $\{$**inv** $p\}$ $W$ $\{\texttt{s} = \texttt{sum(0, n)}\}$ | postcondition weakening 6, 7 |

- (Note I left "(backwards)" out of the assignment rule usages because it's easy to infer that we're using the *wp* of the assignment and postcondition, not the *sp* of the precondition and assignment.)
- To really believe that this is a proof, you have to believe the implication in line 4, which is hard to do unless you expand the substitutions used:
  - $p[\texttt{s+i/s}] \equiv (0 \le \texttt{i} \le \texttt{n} \wedge \texttt{s} = \texttt{sum(0, i)})[\texttt{s+i/s}] \equiv 0 \le \texttt{i} \le \texttt{n} \wedge \texttt{s+i} = \texttt{sum(0, i)}$
  - $p[\texttt{s+i/s}][\texttt{i+1/i}] \equiv (0 \le \texttt{i} \le \texttt{n} \wedge \texttt{s+i} = \texttt{sum(0, i)})\,[\texttt{i+1/i}]$
    $\equiv 0 \le \texttt{i+1} \le \texttt{n} \wedge \texttt{s+i+1} = \texttt{sum(0, i+1)}$
  - This last line is indeed implied by $(p \wedge \texttt{i} < \texttt{n})$, which is $\equiv (0 \le \texttt{i} \le \texttt{n} \wedge \texttt{s} = \texttt{sum(0, i)} \wedge \texttt{i} < \texttt{n})$.

### *G. Why Loop Invariants?*

- One definition for the meaning of a loop uses the states $\tau_0, \tau_1, ..., \tau_k$ that hold at each while loop test, with $\tau_k$ being the first one that satisfies $\neg B$. Each individual $\tau_k$ is describable: $\tau_1 = M(S, \tau_0)$, $\tau_2 \equiv M(S, M(S, \sigma))$, and so on. It turns out that, given a precondition $q$, we can find predicates to describe each $\tau_k$, but in general, we have no guarantee of being able to find a predicate that describes "the first $\tau_k$ that satisfies $\neg B$", where $B$ is the while loop test.
- In general, the loop invariant doesn't give a description of each unique $\tau_k$, it describes a common property that all the $\tau_k$ have. This property is a relationship of the loop variables that is supposed to hold every time the loop test is evaluated.
  - For $\{$**inv** $p\}$ $W$ $\{p \wedge \neg B\}$, we need $p$ to be true when we enter the loop and we need each loop iteration to satisfy $\{p \wedge B\}$ $S$ $\{p\}$. Then when the loop terminates, $p \wedge \neg B$ holds.
  - Semantically, we need our initial state $\tau_0 \vDash p$, and we need the loop body $S$ to satisfy $\tau_m \vDash \{p \wedge B\}$ $S$ $\{p\}$ for all the iterations; the final $\tau$ state satisfies $p \wedge \neg B$, which makes us exit the loop.
- For the loop $\{$**inv** $p\}$ **while** $B$ **do** $S$ **od** $\{p \wedge \neg B\}$, where $W$ is the loop,
  - $p$ is an approximation of the loop's *wp:* $p \rightarrow wp(W, p \wedge \neg B)$.
  - $p \wedge \neg B$ is an approximation of the loop's *sp*: $sp(p, W) \rightarrow p \wedge \neg B$.
  - $p$ also approximates the *wp* and *sp* of the loop body: $p \wedge B \rightarrow wp(S, p)$ and $sp(p \wedge B, S) \rightarrow p$.

 

**Loop preparation and cleanup**

- We often use code to initialize loop variables (variables modified in the loop body) before entering a loop; similarly, we may have code after the loop to clean up any final details once the loop finishes. In general, we might have

$$\{p_0\}\ \textit{initialization code}\ \texttt{;}\ \{\texttt{inv}\ p\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{od;}\ \textit{clean-up code}\ \{q\}$$

- The purpose of loop initialization is to establish the truth of the invariant before the first test of the loop: $\{p_0\}$ *initialization code* $\{p\}$. If there's no initialization code, we just need $p_0 \to p$.
- The purpose of clean-up code is to take us from the loop postcondition to our desired final postcondition $\{p \wedge \neg B\}$ *clean-up code* $\{q\}$. If there's no clean-up code, we just need $p \wedge \neg B \to q$.

### *H. No General wp or sp for Loops*

- In this section, we'll look at the general situations for exact descriptions of the *wp* and *sp* of a loop, to show us why in general they can't be described.
- Let $W \equiv$ **while** $B$ **do** $S$ **od**. For which $\sigma$, $q$, and $r$ do we get $\sigma \vDash \{q\}\ W\ \{r\}$? I.e., if $\sigma \vDash q$, when does $M(W, \sigma) \vDash r$?
- Solving this problem exactly requires us to calculate the *wp* or *sp* of the loop: $q \to wp(W, r)$ or $sp(q, W) \to r$.
- For $wp(W, r)$ (or more specifically $wp(W, r \wedge \neg B)$), let $wp_0$, $wp_1$, … be the weakest preconditions of $W$ and $(r \wedge \neg B)$ when exactly 0, 1, … iterations are required.
  - For zero iterations, we have $\{r \wedge \neg B\}\ W\ \{r \wedge \neg B\}$, so $wp_0 \Leftrightarrow r \wedge \neg B$. For exactly one iteration, we have $wp_1 \Leftrightarrow B \wedge wp(S, wp_0)$: Since $B$ holds, we need to iterate at least once; since $wp(S, wp_0)$ holds, we will terminate after that one iteration.
  - More generally, $wp_{k+1} \Leftrightarrow B \wedge wp(S, wp_k)$, and the full $wp(W, r \wedge \neg B) \Leftrightarrow wp_0 \vee wp_1 \vee \ldots$.
  - Since we don't allow infinite-length predicates, we can only describe $wp(W, r \wedge \neg B)$ exactly if we can find an equivalent finite length predicate, and there's no guarantee one exists. This is why we use an approximation: The invariant $p$ implies each of $wp_0$, $wp_1$, ….
- For $sp(q, W)$, we know $q$ holds after zero iterations; if we terminate, then $q \wedge \neg B$ holds.
  - If we must continue on for iteration 1, we run $W$ starting with $q \wedge B$.
  - More generally, define $sp_0 \equiv q$ and $sp_{k+1} \Leftrightarrow sp(sp_k \wedge B, S)$ so that $sp_k$ is the strongest postcondition of $q$ and $W$ after $k$ iterations under the assumption that at least $k-1$ iterations were necessary.
  - If we terminate after exactly $k$ iterations, then $sp_k \wedge \neg B$ holds.
  - The full $sp(q, W) \Leftrightarrow (sp_0 \wedge \neg B) \cup (sp_1 \wedge \neg B) \cup \ldots$. Factoring out $\neg B$ yields $sp(q, W) \Leftrightarrow \neg B \wedge (sp_0 \vee sp_1 \vee \ldots)$. Again, we have a potentially infinite-length predicate and would need to find a finite-length logical equivalent to give a precise description of $sp(q, W)$, and there's no guarantee that one exists. Instead, we use an approximation: the invariant $p$ is implied by each of $sp_0$, $sp_1$, ….

 

# *Proof Rules for Correctness Triples*

*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 (sequencing, conditional, and iterative).

## *B. Objectives*

At the end of this activity you should

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

## *C. Problems*

In Problems 3 and 5, let "^" mean exponentiation. Use the vertical format to display rule instances.

1. Give the instance of the conditional rule we need to combine $\{x = y \wedge x < 0\}$ `y := -x` $\{y \geq 0\}$ and $\{x = y \wedge x \geq 0\}$ **skip** $\{y \geq 0\}$
2. Our goal is to prove $\{p\}$ **if** `b[M] < x` **then** `L := M` **else** `R := M` **fi** $\{L < R\}$ using *wp*.
   a. Calculate *wp*(`L := M`, `L < R`) and *wp*(`R := M`, `L < R`).
   b. Show the instance of the conditional rule that you get when you use part (a) to build [10/18] the triples. (This is the *wp* of the **if-fi**.)
3. 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.)
4. Consider the incomplete triples $\{p_2\}$ `k := 0` $\{p_1\}$ and $\{p_1\}$ `x :=` $e$ $\{x = 2^k\}$, where $p_1$, and $p_2$ 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.
5. If we want to use the loop rule to prove $\{$**inv** `x = 2^k`$\}$ **while** `k ≠ n` **do** `x := x+x; k := k+1` **od** $\{q\}$
   a. What can we use for $q$?
   b. What triple do we need to prove about the loop body? Show the rule instance.

 

## *Solution to Activities 14 and 15 (Proof Rules)*

[10/18 Made #1 match its question better, did some tweaks to other answers.]

1. (Conditional rule)

   One way to combine $\{x = y \wedge x < 0\}\ y := -x\ \{y \geq 0\}$ and $\{x = y \wedge x \geq 0\}$ **skip** $\{y \geq 0\}$ is to use an **if-then** statement $\{x = y\}$ **if** $x < 0$ **then** $y := -x$ **fi** $\{y \geq 0\}$ (which contains an implicit **else skip**)

   1. $\{x = y \wedge x < 0\}\ y := -x\ \{y \geq 0\}$
   2. $\{x = y \wedge x \geq 0\}$ **skip** $\{y \geq 0\}$
   3. $\{x = y\}$ **if** $x < 0$ **then** $y := -x$ **else skip fi** $\{y \geq 0\}$ conditional 1, 2

   The other way to combine them is to make the **skip** the true branch (this would be pretty weird).

   4. $\{x = y\}$ **if** $x \geq 0$ **then skip else** $y := -x$ **fi** $\{y \geq 0\}$ conditional 2, 1

2. (Prove $\{p\}$ **if** $b[M] < x$ **then** $L := M$ **else** $R := M$ **fi** $\{L < R\}$ using *wp*)

   a. $wp(L := M, L < R) \equiv M < R$ and $wp(R := M, L < M) \equiv L < R$.

   b. The rule instance is

   1. $\{L < M\}\ R := M\ \{L < R\}$
   2. $\{M < R\}\ L := M\ \{L < R\}$
   3. $\{p\}$ **if** $b[M] < x$ **then** $L := M$ **else** $R := M$ **fi** $\{L < R\}$ conditional 1, 2

      where $p \equiv (b[M] < x \rightarrow M < R) \wedge (b[M] \geq x \rightarrow L < M)$

3. (Preconditions for $x = 2\hat{}k$ postcondition)

   a. $p_2 \equiv wp(k := k+1,\ x = 2\hat{}k) \equiv x = 2\hat{}(k+1)$.

   $p_1 \equiv wp(x := x+x,\ p_2) \equiv wp(x := x+x,\ x = 2\hat{}(k+1)) \equiv x+x = 2\hat{}(k+1)$.

   b. The full proof is:

   1. $\{x = 2\hat{}(k+1)\}\ k := k+1\ \{x = 2\hat{}k\}$ assignment
   2. $\{x+x = 2\hat{}(k+1)\}\ x := x+x\ \{x = 2\hat{}(k+1)\}$ assignment
   3. $\{x+x = 2\hat{}(k+1)\}\ x := x+x;\ k := k+1\ \{x = 2\hat{}k\}$ sequence 2, 1

4. (Initially establish $x = 2\hat{}k$)

   a. Let $p_1 \equiv wp(x := e,\ x = 2\hat{}k)) \equiv x = 2\hat{}k)[e/x] \equiv e = 2\hat{}k$ and let $p_2 \equiv wp(k := 0, p_1)$ $\equiv (e = 2\hat{}k)[0/k] \equiv (e[0/k] = 2\hat{}0)$. The simplest value for $e$ is $2\hat{}0$, since then $(e[0/k] = 2\hat{}0) \equiv ((2\hat{}0)[0/k] = 2\hat{}0) \equiv 2\hat{}0 = 2\hat{}0$. But $e \equiv 1$ is more natural,

   b. The full proof is

   1. $\{1 = 2\hat{}0\}\ k := 0\ \{1 = 2\hat{}k\}$ assignment
   2. $\{1 = 2\hat{}k\}\ x := 1\ \{x = 2\hat{}0\}$ assignment
   3. $\{1 = 2\hat{}0\}\ x := 1;\ k := 0\ \{x = 2\hat{}k\}$ seq 2, 1

   We can simplify $1 = 2\hat{}0$ to $T$ by using strengthening on the triple in line 1 or line 3.

 

5. (Powers of 2 loop)
   a. The loop postcondition $q \equiv$ `x = 2^k ∧ k = n` (the invariant and the negation of the test)
   b. The triple we need for the loop body is `{x = 2^k ∧ k ≠ n} x := x+x; k := k+1 {x = 2^k}` (If the invariant and loop test are true, then the loop body re-establishes the invariant.) The rule instance is

      1. `{x = 2^k ∧ k ≠ n} x := x+x; k := k+1 {2^k}`
      2. `{inv x = 2^k}`
         `while k ≠ n do x := x+x; k := k+1 od`
         `{x = 2^k ∧ k = n}` loop

 