Proof-Outline Logic.
(Sequential Programming Edition)

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

Draft typeset January 10, 2020

Abstract

An introduction to proof outlines and Hoare logic.

Contents

Note on editions: This is the Sequential Programming edition, designed to support MUN course Engi-5892 Algorithms: Correctness and Complexity. There is also a Concurrent Programming edition, which contains additional sections on concurrent programming. The Concurrent Programming edition uses a somewhat different notation —one that better matches Andrews's text book. [Andrews, 2000].

0 Preface

This note provides background on assertions and the use of assertions in designing correct programs.

The ideas presented are mainly due, for the sequential programming part, to Floyd [Floyd, 1967] and Hoare [Hoare, 1969]. Blikle [Blikle, 1979] presented an early version of proof-outline logic.

Sections 1 and 2 deal with sequential programming. As such they are an elaboration of Hoare's excellent 'Axiomatic basis' paper [Hoare, 1969]. I suggest reading Hoare's paper first. For sequential programs, proof-outline logic is just like Hoare logic except that commands contain internal assertions.

1 Assertions and logic

1.0 Conditions and assertions

A **condition** is a boolean expression with free variables chosen from the state variables of a program. For example, if variables x and y of type int appear in a program, then the following are all examples of conditions

$$\begin{aligned} x &< y \\ x &= y \\ x + y &= 0 \\ x &\geq 0 \\ x &\geq 0 \wedge x + y = 0 \qquad . \end{aligned}$$

A condition that is expected to be true every time execution passes a particular point in a program is called an **assertion**. In this course, assertions are preceded by ## and are followed by an end-of-line like this:[0]

```
int x ;
int y ;
x := 5 ;
y := -5 ;
## $x \geq 0 \wedge x + y = 0$
z := x+z ;
```

Sometimes I'll write assertions inside curly brackets like this:

```
int x ; int y ; x := 5 ; y := -5 ; {$x \geq 0 \wedge x + y = 0$} z := x+y ;
```

[0]This fragment uses both the symbol := and =.This is one of those "notational improvements" I mentioned. I will use := for assignment and either = or == for equality when writing pseudo-code. In C, C++, and Java, = is used for assignment and == for equality. Andrews follows the C/C++/Java convention. If you want to be on the safe side of any possible misunderstanding, you can use := for assignment and == for equality.

1.1 Assertions in C, C++, and Java

If one is programming in C or C++, then assertions may be written either as comments or using the assert macro from the standard C library. E.g.

```
#include <assert.h>
...
int x ;
int y ;
x = 5 ;
y = -5 ;
assert( x >=0 && x+y == 0 ) ;
```

Assertions written using the assert macro will be evaluated at run time and the program will come to a grinding halt, should the assertion ever evaluate to false.[1]

In Java one can easily create one's own Assert class with a check method in it.[2]

```
public class Assert {
    public static void check( boolean b ) {
        if( !b ) { throw new java.lang.AssertionError() ; }
    }
}
```

This can be used in your code as follows:

```
int x ;
int y ;
x = 5 ;
y = -5 ;
Assert.check( x >=0 && x+y == 0 ) ;
```

1.1.0 Be an assertive programmer

Using assertions has several benefits.

- In the design process, they help you articulate what conditions you expect to be true at various points in program execution.
- In testing, executable assertions can help you identify errors in your code or in your design.

[1]By using a different include file, one can, of course, make the action followed on a false assertion be whatever you like. For example, in a desktop application, one might cause all files to be saved and an error report to be assembled and e-mailed back to the developers; in an embedded system, one might cause the system to go into safe mode. If you program in C or C++, I strongly suggest redefining the assert macro in a way that suits your application. And use it!

[2]As of Java 1.4, there is actually an assert keyword in Java. However I don't recommend its use. Assertion checking is turned off by default in most (if not all) JVMs. This can be compared to removing the seat belts from a car's design once it goes into production. In my own work I use my own assertion checking class. I recommend you do the same.

- In execution, executable assertions —combined with a recovery mechanism— can help make your program more fault tolerant.
- Assertions provide valuable documentation. Executable assertions are more valuable than comments, as they are more likely to be accurate.

Whether to code assertions as comments or as executable checks is a question that depends on the local conditions of the project you are working on. Sometimes it has to be answered on a case-by-case basis. In this course we will concentrate on the use of assertions in the design process, rather than on their (nevertheless important) uses in testing, documentation, and in making systems fault-tolerant. My general advice is to make assertions executable as much as is practical.[3]

1.2 Substitutions

Sometimes it is useful to create a new condition by replacing all free occurrences of a variable x in a condition P by an expression (E). We write $P[x : E]$ for the new condition.[4] For example

$$\begin{array}{lll}(x \geq 0 \wedge x + y = 0)\,[x : z] & \text{is} & (z) \geq 0 \wedge (z) + y = 0 \\ (x \geq 0 \wedge x + y = 0)\,[x : x + y] & \text{is} & (x + y) \geq 0 \wedge (x + y) + y = 0 \\ (2y = 5)\,[y : y + z] & \text{is} & 2(y + z) = 5 \qquad .\end{array}$$

It is useful to extend this notation to allow simultaneous substitution for more than one variable. For example

$$(x \geq 0 \wedge x + y = 0)\,[x, y : z, x] \text{ is } (z) \geq 0 \wedge (z) + (x) = 0$$

Usually we omit the parentheses in contexts where they are not required.

1.3 Propositional and predicate logic

In this section, I will review a little bit of propositional and predicate logic. We use notations $\neg$ (not), $=$ (equality), $\wedge$ (and), $\vee$, (or), and $\Rightarrow$ (implication). Precedence between the operators

[3]In concurrent programming there is an additional complication in making assertions executable, namely that they should be evaluated atomically. Consider the assertion

$$x = 0 \vee y = 0$$

if we evaluate this in parallel with the following sequence of assignments

$$x := 0; y := 1;$$

it is possible that the assertion will evaluate to false even though there is no time at which it is in fact false.

This problem can be solved by evaluating assertions only when the thread has exclusive access to the data they refer to.

[4]A variety of notations are used by authors for substitutions. I like this one because it doesn't use subscripts or superscripts.

is in the same order. Some of the laws of propositional logic that will be useful are

$(P \Rightarrow Q) = (\neg P \vee Q)$	Material implication
$(\mathit{true} \Rightarrow P) = P$	Identity
$\mathit{false} \Rightarrow P$	Antidomination
$P \Rightarrow P$	Reflexivity
$P \Rightarrow \mathit{true}$	Domination
$(P \Rightarrow (Q \Rightarrow R)) = (P \wedge Q \Rightarrow R)$	Shunting
$(P0 \Rightarrow Q) \Rightarrow (P0 \wedge P1 \Rightarrow Q)$	Subsetting the antecedent
$(P \Rightarrow Q \wedge R) = (P \Rightarrow Q) \wedge (P \Rightarrow R)$	Distributivity

Also frequently useful are the **one-point laws**, which let you make use of information from equalities. E and F range over expressions of any type, v is a variable of that same type.

$$(E = F \Rightarrow P[v : E]) = (E = F \Rightarrow P[v : F])$$
$$(E = F \wedge P[v : E]) = (E = F \wedge P[v : F])$$

To see that these are true, consider the case where $E = F$ and then the case where $E \neq F$.

For example
$$x = X \wedge y = Y \Rightarrow x^y = X^Y$$
simplifies, using one-point (and shunting), to
$$x = X \wedge y = Y \Rightarrow X^Y = X^Y$$
which then simplifies to
$$x = X \wedge y = Y \Rightarrow \mathit{true}$$
which is then true.

Any condition that is true for all assignments of values to its free variables, is called a **universally true** formula. For example, (assuming x and y and z are integer variables) the following are all universally true

$$2 + 2 = 4$$
$$x < y \wedge y < z \Rightarrow x < z$$
$$x + 1 > x$$

However, $x^2 + y^2 = z^2$ is not universally true, because there is an assignment for which it is not true; for example $3^2 + 4^2 = 6^2$ is not true.

Whether a condition is universally true may depend on the types ascribed to its variables. For example, if we are using Java and x has type **int**, then, by the rules of the Java language, the value of $x + 1$, when x is $2^{31} - 1$, is -2^{31}; so $x + 1 > x$ is not universally true when x represents a Java **int** and $+$ is interpreted as Java **int** addition.

Sometimes expressions are undefined, for example, supposing x and y are rational variables, x/y is undefined when y is 0. This raises the question of whether $1 = x/x$ is universally true. We'll say that such an expression is not universally true because sometimes it is undefined. However, the expression $x \neq 0 \Rightarrow 1 = x/x$ is universally true; if we consider the case of $x = 0$, we have false $\Rightarrow$?, and, applying the principle of antidomination, this is true whether the ? represents something true or false.

We will follow the C, Java, C++ and Dafny languages by considering the Boolean operators $\wedge$, $\vee$, and $\Rightarrow$ as being undefined if (and only if) their left operand is undefined or if their left operand is defined, their right operand is undefined, and the value of the left operand does not determine a value. Using ? to represent an unknown or undefined truth value, we can fill in truth tables for the propositional logic as follows

$\neg$	
false	true
?	?
true	false

$\wedge$	false	?	true
false	false	false	false
?	?	?	?
true	false	?	true

$\vee$	false	?	true
false	?	?	true
?	?	?	true
true	true	true	true

$\Rightarrow$	false	?	true
false	true	true	true
?	?	?	?
true	false	?	true

Note that, in this logic, the "and" and "or" operators are not necessarily commutative. Before applying the commutative laws, we should convince ourselves that both operands are defined. Furthermore the identity, reflexivity, and domination laws for implication only hold when it is known that P is defined.

2 Sequential programming

2.0 Contracts

A specification for a component indicates the operating conditions (i.e. the conditions under which the component is expected to operate) and the function of the component (i.e. the relationship between the component's inputs and outputs). For example we might specify a resistor by saying that the relationship between the voltage and current across the resistor is given by

$$953I \le V \le 1050I \qquad ,$$

provided

$$0 \le V \le +10 \qquad .$$

The latter formula gives the operating conditions, the former the relation between inputs and outputs.

In programming, we can use a pair of assertions as a specification or "contract" for a command or subroutine. The first assertion is the so-called precondition, it specifies the operating conditions, that is, the state of the program when the command begins operation. The second assertion specifies the state of the program when (and if) the command ends operation. For example, the following pair of conditions specifies a command that results in x being assigned the value 5, provided that y is initially 4

$$[y = 4, x = 5] \qquad ,$$

where x and y are understood to be state variables of type `int`. One solution to this particular contract is

$\{y = 4\}$
$x := y + 1$
$\{x = 5\}$.

Such a triple, consisting of a precondition, a command, and a postcondition, is called a **Hoare triple** after C.A.R. Hoare, who introduced the idea to programming. Here is another solution:

$\{y = 4\}$
$x := 5$;
$\{x = 5\}$.

Here is one more:

$\{y = 4\}$
$y := y + 1$;
$x := y$;
$\{x = 5\}$.

Nothing in the contract says that y must not change!

If you do want to specify that a variable does not change, then 'constants' can be used. Constants are conventionally written with capital letters. The following contract specifies that, provided y is initially less than 100, y must not change and the final value of x must be larger than that of y:

$$[y < 100 \wedge y = Y, y = Y \wedge x > y]$$

where it is understood that x and y are state variables of type int, while Y is a constant[5] of type int.

2.1 Partial correctness

Consider a Hoare triple $\{P\}\ S\ \{Q\}$. We define that the triple is **partially correct** if and only if, for all possible values of all constants, whenever the execution of S is started in a state satisfying P, the execution of S does not crash and can only end in a state satisfying Q.[6]

[5]The word 'constant' is the traditional term to use. In the mathematical sense, Y is a variable. We use the term 'constant' to distinguish such mathematical variables from "program variables" which refer to components of the program state. The point is that Y can't be changed by the execution of the program so Y represents the same value in both the precondition and in the postcondition. Since the precondition implies $y = Y$ and the postcondition implies $y = Y$, it is clear that y has the same value in the final state as it has in the initial state.

[6]This definition is incomplete because it assumes that the reader has an understanding of what it means for a program to execute starting from a given state. This can be formalized mathematically by giving a "semantics" to our programming language.

Such a semantics would define, for each statement S, a boolean function s so that $s(\sigma, \sigma')$ is true if and only if a computation executing S, started in state σ, could terminate in a state σ'. (I say *could* terminate, to allow for nondeterminism.) We use a special state $\perp$ to mean that the computation has crashed, so $s(\sigma, \perp)$ means that a computation executing S,started in state σ, could terminate by crashing.

Let's suppose that the expression P corresponds to a boolean function p such that $p(\sigma, x_1, x_2, \ldots x_k)$ is true only when P describes state σ and constants x_1, x_2, ..., and x_k; and that Q corresponds similarly to a boolean function q. We assume that p and q are false when the state is $\perp$.

We can define 'partially correct' in a more formal way by saying that $\{P\}\,S\,\{Q\}$ is partially correct if and only if, for all states σ and σ', and for all values x_1, x_2, ..., and x_k, if $p(\sigma, x_1, x_2, \ldots x_k)$ and $s(\sigma, \sigma')$, then

Note that the definition of partial correctness does not require that S should terminate. Thus the following triple is partially correct even if we interpret x to have the type $\mathbb{Z}$, that is, to range over all mathematical integers

$$\{\text{true}\} \quad \textbf{while } x \neq 0 \textbf{ do } x := x - 1 \textbf{ end while} \quad \{x = 0\} \qquad .$$

If we consider initial states where x is positive or zero, then eventually the while-loop will terminate and the program will halt in a state where $x = 0$, satisfying the postcondition. If we start the while-loop in an initial state where x is negative, then the while-loop will never terminate and so execution "can only end in a state satisfying" the postcondition by virtue of the fact it never ends at all!

In concurrent programming, we are often interested in processes that do not terminate (e.g., in embedded systems) so dealing with partial correctness is an appropriate and desirable thing to do. If termination is important, we can deal with it as a separate concern. From here on, we won't be worried about any other kind of correctness, so we will just say "correct".

2.2 Some examples of assignments and a rule

Here are some small examples of Hoare triples. In each case the variables should be understood to be integers

$$\{x + 1 = y\}\ x := x + 1\ \{x = y\}$$

Is this triple correct? (Answer for yourself before reading on...) If initially y is $x + 1$ and we change x to $x + 1$, then finally both x and y will equal the original value of $x + 1$, and so they will equal each other. Yes, it is correct.

How about

$$\{2x = 3y\}\ x := 2x\ \{x = 3y\}$$

Is this correct? Well, if initially $2x$ is $3y$, then, after changing x to $2x$, finally x will be $3y$.

These two examples suggest a general rule, which is: for any condition Q, variable v, and expression E,

$$\{Q[v : E]\}\ \ v := E\ \ \{Q\}\ \text{ is correct.}$$

When Q is the postcondition of an assignment $v := E$, we call $Q[v : E]$ the **substituted postcondition**.

The rule above is sound, but it does not let us show that

$$\{2x < 3y\}\ \ x := 2x\ \ \{x \leq 3y\}$$

is correct. The substituted postcondition is $(x \leq 3y)[x : 2x]$, which is $2x \leq 3y$, but the precondition is not that, it is $2x < 3y$. This motivates a more general rule: for any conditions P and Q, variable v, and expression E,

$$\{P\}\ \ v := E\ \ \{Q\}\ \text{ is correct if } P \Rightarrow Q[v : E] \text{ is universally true.}$$

There is one more aspect to assignment that should be mentioned. This is that the expression might not always be defined. For example, if we divide by 0, this may be an error

$q(\sigma', x_1, x_2, \ldots x_k)$.

This definition is still incomplete since I haven't defined how a function s is derived from each statement S, but I'll leave that as an exercise.

and we should consider that, if this happens, the program has crashed.[7] Since a command that crashes is not partially correct, we should really ensure that our rule for assignments includes checking that the expression is well defined. Let's suppose that, for each expression E, there is a condition $\mathrm{df}[E]$ that says that E is well-defined, i.e. does not crash when evaluated. For example, $\mathrm{df}[x/y]$ might be $y \neq 0$. Now the improved assignment rule is

$$\{P\}\ v := E\ \{Q\} \text{ is correct} \quad \begin{array}{l} \text{if } P \Rightarrow \mathrm{df}[E] \text{ is universally true} \\ \text{and } P \Rightarrow Q[v : E] \text{ is universally true} \end{array} \qquad \text{(assignment rule)}$$

In many cases $\mathrm{df}[E]$ is simply true, and so it is trivial that $P \Rightarrow \mathrm{df}[E]$ is universally true.

This rule generalizes to simultaneous assignments to multiple variables. For two variables it is

$$\{P\}\ v, w := E, F\ \{Q\} \text{ is correct} \quad \begin{array}{l} \text{if } P \Rightarrow \mathrm{df}[E] \wedge \mathrm{df}[F] \text{ is universally true} \\ \text{and } P \Rightarrow Q[v, w : E, F] \text{ is universally true} \end{array} \qquad \text{(assignment rule)}$$

For example

$$\{x < y\}\ x, y := y, x\ \{y \leq x\}$$

The substituted postcondition is $(y \leq x)\,[x, y : y, x]$, which is $x \leq y$; this is implied (for all values of x and y) by $x < y$.

Here is one last example of an assignment; it will be of use later.

$$\{y \geq 0 \wedge x = X \wedge y = Y\}\ z := 1\ \left\{y \geq 0 \wedge X^Y = z \times x^y\right\}$$

First we find the substituted postcondition

$$y \geq 0 \wedge X^Y = 1 \times x^y$$

which simplifies to

$$y \geq 0 \wedge X^Y = x^y$$

This (using one-point laws) is implied by the precondition $y \geq 0 \wedge x = X \wedge y = Y$.

2.3 A bigger example

Here is another example. I claim that

$$\{y \geq 0 \wedge x = X \wedge y = Y\}\ S\ \left\{z = X^Y\right\} \qquad , \tag{0}$$

is correct, where

$$\begin{aligned} S &\triangleq (z := 1; \textbf{while}(\ y > 0\ )\ T) \\ T &\triangleq \textbf{if } \mathrm{odd}(y) \textbf{ then } U \textbf{ else } V \textbf{ end if} \\ U &\triangleq (z := z \times x; y := y - 1;) \\ V &\triangleq (x := x \times x; y := y \text{ div } 2;) \qquad . \end{aligned}$$

Here all variables are integers. y div 2 means the integer part of $y/2$, so if y is even, $y \text{ div } 2 = \frac{y}{2}$.

To show that this triple is correct, we'll need to deal with constructs other than assignments. For that we introduce a new idea: proof outlines.

[7]This particular example depends on the language. In C and C++, dividing by 0 is considered 'undefined behaviour', which means anything can happen. In this case, we might as well assume the worst possible outcome. For the purpose of this essay, we'll assume this is crashing the program. In Java, on the other-hand, a floating point division by 0 is perfectly well defined.

```
{$y \geq 0 \wedge x = X \wedge y = Y$}
$z := 1$ ;
{$y \geq 0 \wedge X^Y = z \times x^y$}
while $y > 0$ do
    {$X^Y = z \times x^y \wedge y > 0$}
    if $odd(y)$ then
        {$X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{odd}(y)$}
        $z := z \times y$
        {$y - 1 \geq 0 \wedge X^Y = z \times x^{y-1}$}
        $y := y - 1$
    else
        {$X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{even}(y)$}
        $x := x \times x$
        {$\mathrm{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{\frac{y}{2}}$}
        $y := y$ div 2
    end if
end while
{$z = X^Y$}
```

Figure 0: An example proof outline.

2.4 Proof outlines

A **proof outline** is a command that is annotated with assertions. It represents the outline of a proof of the program. Figure 0 is a proof outline for the example of the last section.

A proof outline is not a proof; it is a summary of a proof. This is why it is called a 'proof outline'.

2.5 Correctness of proof outlines

2.5.0 Definition

We define that the proof outline $\{P\}\ S\ \{Q\}$ is **partially correct** if and only if, for all possible values of all constants, whenever the execution of S is started in a state satisfying P, the execution of S does not crash and can only end in a state satisfying Q and that each time an internal assertion is encountered, it is true. Note that an assertion that comes right before a loop must be true each time the loop condition is checked.

Suppose $\{P\}\ S\ \{Q\}$ is a partially correct proof outline. Let $\widehat{S}$ be formed by deleting all assertions from S or by treating them as comments. Now $\{P\}\ \widehat{S}\ \{Q\}$ is a partially correct Hoare triple.

2.5.1 Rules

Earlier we saw a rule for assignment; here I'll give rules for all proof outlines.

Assignment Rule: $\{P\}\ v := E\ \{Q\}$ is a partially correct proof outline if

$$P \Rightarrow Q[v : E] \text{ is universally true and } P \Rightarrow \mathrm{df}[E] \text{ is universally true} \qquad .$$

Skip Rule: $\{P\}$ **skip** $\{Q\}$ is a partially correct proof outline if

$$P \Rightarrow Q \text{ is universally true} \qquad .$$

(Sequential) Composition Rule: $\{P\}\ S\ \{Q\}\ T\ \{R\}$ is a partially correct proof outline, provided

- that $\{P\}\ S\ \{Q\}$ is a partially correct proof outline, and
- that $\{Q\}\ T\ \{R\}$ is a partially correct proof outline.

2-Tailed If Rule: $\{P\}$ **if**(E) $\{Q_0\}\ S$ **else** $\{Q_1\}\ T\ \{R\}$ is a partially correct proof outline, provided

- that $\{Q_0\}\ S\ \{R\}$ is a partially correct proof outline,
- that $\{Q_1\}\ T\ \{R\}$ is a partially correct proof outline,
- that $P \Rightarrow \mathrm{df}[E]$ is universally true,
- that $P \wedge E \Rightarrow Q_0$ is universally true, and
- that $P \wedge \neg E \Rightarrow Q_1$ is universally true.

1-Tailed If Rule: $\{P\}$ **if**(E) $\{Q\}\ S\ \{R\}$ is a partially correct proof outline, provided

- that $\{Q\}\ S\ \{R\}$ is a partially correct proof outline,
- that $P \Rightarrow \mathrm{df}[E]$ is universally true,
- that $P \wedge E \Rightarrow Q$ is universally true, and
- that $P \wedge \neg E \Rightarrow R$ is universally true.

Iteration Rule: $\{P\}$ **while**(E) $\{Q\}\ S\ \{R\}$ is a partially correct proof outline, provided

- that $P \Rightarrow \mathrm{df}[E]$ is universally true,
- that $P \wedge E \Rightarrow Q$ is universally true,
- that $P \wedge \neg E \Rightarrow R$ is universally true, and
- that $\{Q\}\ S\ \{P\}$ is a partially correct proof outline.

By the way, the loop's precondition, P, is called an **invariant** of the loop. Loop invariants are crucial in designing and documenting loops. Note that, provided it is true when the while command starts, the invariant will be true at the start of each iteration and when the loop terminates. Loop invariants allow us to analyze the effect of a loop by considering only the effect of a single iteration.

From here on we will write "**correct**" in place of "partially correct", as we won't be concerned with any other sort of correctness.

In a proof outline, all commands will be preceded by an assertion. This is its **precondition**. In the example, the precondition of $y := y/2;$ is

$$\mathrm{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{y/2}$$

and the precondition of $x := x \times x;$ is

$$X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{even}(y) \qquad .$$

A proof outline is **provably correct** if it can be shown to by correct using just the rules of this section (plus whatever rules or techniques are needed to show that conditions are universally true. Not all correct proof outlines are provably correct. For example

$$\{x = 1\}\ x := x + 1\ \{x > 0\}\ x := x + 1\ \{x = 3\}$$

is correct, but it is not provably correct.

2.6 Correctness of the example

There is a little, but not much, work left to show that the example proof outline in Figure 0 is correct. First a recall that a boolean formula is said to be universally true if it evaluates to true regardless of the values chosen for its free variables (including our so-called constants).

Let's call the loop invariant I.

$$I \triangleq y \geq 0 \wedge X^Y = z \times x^y$$

- Because of the first assignment, we must show

  $$y \geq 0 \wedge x = X \wedge y = Y \Rightarrow I[z : 1]$$

  is universally true. After substitution we have

  $$y \geq 0 \wedge x = X \wedge y = Y \Rightarrow y \geq 0 \wedge X^Y = 1 \times x^y$$

  which (using a one-point law) we can easily see is universally true.

- From the rule for while-loops, we must show

  $$I \wedge y > 0 \Rightarrow X^Y = z \times x^y \wedge y > 0$$
  $$I \wedge \neg (y > 0) \Rightarrow z = X^Y$$

  are each universally true. Both are fairly straight-forward.

- From the rule for 2-tailed if commands, we must show

$$X^Y = z \times x^y \wedge y > 0 \wedge \text{odd}(y) \Rightarrow X^Y = z \times x^y \wedge y > 0 \wedge \text{odd}(y) \text{ and}$$
$$X^Y = z \times x^y \wedge y > 0 \wedge \neg\text{odd}(y) \Rightarrow X^Y = z \times x^y \wedge y > 0 \wedge \text{even}(y)$$

  are each universally true. Both are trivial.

- The four assignments in the loop body give rise to four expressions that should be shown to be universally true:

$$X^Y = z \times x^y \wedge y > 0 \wedge \text{odd}(y) \Rightarrow \left(y - 1 \geq 0 \wedge X^Y = z \times x^{y-1}\right)[z : z \times y]$$
$$y - 1 \geq 0 \wedge X^Y = z \times x^{y-1} \Rightarrow I[y : y - 1]$$
$$X^Y = z \times x^y \wedge y > 0 \wedge \text{even}(y) \Rightarrow \left(\text{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{\frac{y}{2}}\right)[x : x \times x]$$
$$\text{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{\frac{y}{2}} \Rightarrow I[y : y \text{ div } 2]$$

  Each of these can be shown to be true using the usual laws of arithmetic.

References

[Andrews, 2000] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and distributed programming.* Addison Wesley Longman, 2000.

[Blikle, 1979] Andrzej J. Blikle. Assertion programming. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, number 74 in Lecture Notes in Computer Science, pages 26–42, 1979.

[Floyd, 1967] Robert Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, Volume XIX*, 1967.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.