

Chapter 2

The Axiomatic Approach

An axiomatic approach requires both axioms and rules of inference. For our purposes, we will accept the axioms of arithmetic which characterize the integers and the usual rules of logical inference as discussed earlier (Chapter 1, Section 5). However, we do not seek to model just the ‘world of integers’ but also the ‘world of computation,’ and hence we need additional axioms and inference rules corresponding to the constructs we intend to use as our programming language.

1. PROGRAMMING LANGUAGE AXIOMS

In order to build programs into our inference system, we must be able to model a computation as a logical predicate. We adopt the notation $\{P\}S\{Q\}$, where P and Q are assertions about the state of the computation and S is a statement. This expression is interpreted as “If P , called the **precondition**, is true before executing S and S terminates normally, then Q , called the **postcondition**, will be true.” This will allow us to model a simple “Algol-like” programming language. We do this by adding the inference rules of composition and consequence:

$$\text{Composition : } \frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}}$$

$$\begin{array}{l} \text{Consequence}_1 : \frac{\{P\}S\{R\}, R \Rightarrow Q}{\{P\}S\{Q\}} \\ \text{Consequence}_2 : \frac{P \Rightarrow R, \{R\}S\{Q\}}{\{P\}S\{Q\}} \end{array}$$

If we are to have confidence in a program which has been proven to be partially correct by this axiomatic method, then it is important for us to believe in the axioms and inference rules accepted up front. A good way to find motivation for the choice of these inference rules is to examine the Floyd-style flow chart associated with each drawing.

The rule of composition is the basic mechanism which allows us to “glue” together two computations (i.e., two statements). If the postcondition of one statement is the same as the precondition of the following statement, then the execution of both of them proceeds as expected.

The rules of consequence build in our predicate logic rules of inference. We can use these to permit us to use alternative pre- and postconditions to any statement as long as we can derive them via our mathematical system.

Given this logical model, we need to build in the semantics of a given programming language. We will use the following BNF to describe a simple Algol-like language:

```

< stmt > ::= < stmt >; < stmt >
           | if < expr > then < stmt >
           | if < expr > then < stmt > else < stmt >
           | while < expr > do < stmt >
           | < id > := < expr >

```

where < id > and < expr > have their usual meanings. Note: The examples will be kept simple, and we will not worry about potential ambiguities such as programs like: < stmt >; **if** < expr > **then** < stmt >; < stmt >. that is, is this a “statement and an **if**” or “statement, **if**, and statement?” The examples will be clear as to meaning.

The basic approach is a “backwards verification” method. For example, for the assignment, we will define the axiom so that given the result of an assignment statement, what must have been true prior to the execution of the statement in order for the result to occur? Or in other words, given the postcondition to an assignment statement, what is its precondition?

We accept the assignment axiom *schema*

$$\{P_y^x\}x := y\{P\}$$

Here P_y^x represents the expression P where all free occurrences of x have been replaced by y . For example,

$$\{z = x + y - 3\}x := x + y + 1\{z = x - 4\}$$

represents the effect of replacement of x by $x + y + 1$.

In addition to the above axiom schema, we accept the following rules of inference for the **if** and **while** statements:

$$\begin{array}{l} \text{Conditional}_1 : \frac{\{P \wedge B\} S \{Q\}, P \wedge \neg B \Rightarrow Q}{\{P\} \text{if } B \text{ then } S\{Q\}} \\ \text{Conditional}_2 : \frac{\{P \wedge B\} S_1 \{Q\}, \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{if } B \text{ then } S_1 \text{ else } S_2 \{Q\}} \\ \text{While} : \frac{\{P \wedge B\} S \{P\}}{\{P\} \text{while } B \text{ do } S \{P \wedge \neg B\}} \end{array}$$

The **if** statement rules are fairly obvious. Each of the two possible rules simply traces the execution of the corresponding *if – then* and *if – then – else* statement, and given precondition P , determines under what conditions postcondition Q will be true.

The axiom for the **while** statement is perhaps as major a contribution by Hoare [29] as the entire model of verification that he also developed. The basic loop has the structure:

while *something – is – true*
 do *something*

There must be some property that remains true throughout the execution of the loop. So if we call the body of the loop S , then we will call the property that must remain true P , and we have the result: “If P is

true and we execute the loop, then P will remain true.” The condition ‘and we execute the loop’ is just the predicate on the **while** statement B . This leads to the condition that: $\{P \wedge B\}S\{P\}$ as the antecedent property on the **while** axiom. So if this antecedent is true, then after the loop ‘terminates,’ we still will have P true, and since the loop terminated, B must now be false, hence the axiom as given. Note we have not proven that the loop *does* terminate. That must be shown and is outside of this axiom system (see Section 1.2).

The property that remains true within the loop is called the *invariant* and it is at the heart of developing axiomatic proofs for programs. While there is no algorithmic method for developing invariants, we later give some ideas on how to proceed.

1.1. Example: Integer Division

Consider the following example of a Hoare-style proof of partial correctness of a program to compute the remainder and quotient from a division of two integers, x and y :

$$\begin{aligned} \text{PROGRAM} \equiv \{ \\ & q := 0; \\ & r := x; \\ & \text{while } y \leq r \text{ do} \\ & \quad r := r - y; \\ & \quad q := 1 + q \} \end{aligned}$$

Our input condition is $\{x \geq 0 \wedge y > 0\}$, and our desired output condition is $\{\neg(y \leq r) \wedge x = r + yq\}$. Note how the output condition characterizes the desired relationship between values in order for r and q to represent the remainder and quotient, respectively. Also note that the “input” to this program is assumed to be whatever is contained in the variables upon starting execution of this program.

If our output condition is to be true, it must be so as a result of application of a **while** rule of inference, corresponding to the execution of the only **while** loop in our program. In order to apply the rule, we must identify the P and B in the rule’s antecedent “ $P \wedge B$.” This immediately suggests that our B must be $B \equiv r \geq y$, and that our P (which is referred to as the *invariant*) must therefore be $P \equiv x = yq + r \wedge 0 \leq r$. Hence the inference rule which could be applied would be

$$\frac{\{P \wedge r \geq y\} r := r - y; q := q + 1 \{P\}}{\{P\} \text{ while } r \geq y \text{ do } r := r - y; q := q + 1 \{P \wedge \neg B\}}$$

Continuing to reason backwards from our target assertion, we try to show how the antecedent to this **while** rule would be derived, that is, we must show that the assertion which characterizes the program state just before execution of the **while** loop is maintained by continued execution of the while loop. In short, this inductive step is called “showing that the invariant is maintained.” We start with our invariant P and work backwards through the body of our while loop to see what conditions before the loop will imply P : First, our assignment axiom schema can be brought in, yielding

$$\{x = (q + 1)y + r \wedge 0 \leq r\} q := q + 1 \{P\}$$

and an additional use of the axiom shows us

$$\underbrace{\{x = (q + 1)y + (r - y) \wedge 0 \leq r - y\} r := r - y}_{x = qy + r \wedge 0 \leq r - y} \\ \{x = (q + 1)y + r \wedge 0 \leq r\}$$

By use of our rule of composition applied to the above two steps, we get

$$\frac{\{x = qy + r \wedge 0 \leq r - y\} r := r - y \{x = (q + 1)y + r \wedge 0 \leq r\}, \\ \{x = (q + 1)y + r \wedge 0 \leq r\} q := q + 1 \{P\}}{\{x = qy + r \wedge 0 \leq r - y\} r := r - y; q := q + 1 \{P\}}$$

Now by using a rule of consequence, we can show that the invariant is indeed maintained:

$$\frac{\{x = qy + r \wedge 0 \leq r - y\} r := r - y; q := q + 1 \{P\}, \\ P \wedge r \geq y \Rightarrow x = qy + r \wedge 0 \leq r - y}{\{P \wedge r \geq y\} r := r - y; q := q + 1 \{P\}}$$

Now we must determine whether or not the “initialization” steps in our program will yield us the invariant P . Again working backwards, we utilize our assignment axiom

$$\{x = qy + x \wedge 0 \leq x\} r := x \{P\}$$

utilize it again for

$$\underbrace{\{x = 0y + x \wedge 0 \leq x\}}_{(x=x \wedge 0 \leq x) \equiv 0 \leq x} q := 0 \{x = qy + x \wedge 0 \leq x\}$$

and so by using our rule of composition, we may infer

$$\frac{\{0 \leq x\} q := 0 \{x = qy + x \wedge 0 \leq x\}, \{x = qy + x \wedge 0 \leq x\} r := x \{P\}}{\{x = x \wedge 0 \leq x\} q := 0; r := x \{P\}}$$

Use composition to add the loop initialization to the loop body:

$$\frac{\{0 \leq x\} q := 0; r := x \{P\}, \{P\} \mathbf{while} \ r \geq y \ \mathbf{do} \ r := r - y; q := q + 1 \{P \wedge \neg B\}}{\{0 \leq x\} q := 0; r := x; \mathbf{while} \ r \geq y \ \mathbf{do} \ \dots \{P \wedge \neg B\}}$$

The input assertion implies the antecedent of our initialization code, and so we may use a rule of consequence to replace $0 \leq x$ with the input assertion:

$$\frac{\{0 \leq x\} q := 0; r := x; \mathbf{while} \ r \geq y \ \mathbf{do} \ \dots \{P \wedge \neg B\}, \{x \geq 0 \wedge y > 0 \Rightarrow 0 \leq x\}}{\{x \geq 0 \wedge y > 0\} q := 0; r := x; \mathbf{while} \ r \geq y \ \mathbf{do} \ \dots \{P \wedge \neg B\}}$$

Since $P \wedge \neg B$ is our desired output assertion, we have completed a demonstration of partial correctness for this program.

The program generates two values: q and r . We can use these to have this program model integer division (quotient or *quot*) and the remainder function (*rem* or *mod*) as follows:¹

$$\begin{aligned} (\mathit{quot}(x, y) = q) &\triangleq (\exists r \mid y > r \geq 0 \wedge x = r + yq) \\ (\mathit{rem}(x, y) = r) &\triangleq (\exists q \mid y > r \geq 0 \wedge x = r + yq) \end{aligned}$$

¹This is left as an exercise.

We will use these results later.

The example presented is organized to give as much insight as possible as to *how* each step is chosen. As with so many proofs, we start with the desired result and “reason back” to find out what we needed to start with. This is fine. However, it is also useful to consider the “straightforward” proof as well, as demonstrated by Hoare [29] in his original CACM paper. The above discussion can be organized into a precise proof of our program’s partial correctness as follows:

- | | | |
|-----|--|--------------------|
| 1. | true $\Rightarrow x = x + y \times 0$ | lemma ² |
| 2. | $\{x = x + y \times 0\} r := x \{x = r + y \times 0\}$ | assignment |
| 3. | $\{x = r + y \times 0\} q := 0 \{x = r + yq\}$ | assignment |
| 4. | $\{\mathbf{true}\} r := x \{x = r + y \times 0\}$ | consequence(1,2) |
| 5. | $\{\mathbf{true}\} r := x; q := 0 \{x = r + yq\}$ | composition(4,3) |
| 6. | $x = r + yq \wedge y \leq r \Rightarrow x = (r - y) + y(1 + q)$ | lemma ² |
| 7. | $\{x = (r - y) + y(1 + q)\} r := r - y$
$\{x = r + y(1 + q)\}$ | assignment |
| 8. | $\{x = r + y(1 + q)\} q := 1 + q \{x = r + yq\}$ | assignment |
| 9. | $\{x = (r - y) + y(1 + q)\} r := r - y; q := 1 + q$
$\{x = r + yq\}$ | composition(7,8) |
| 10. | $\{x = r + yq \wedge y \leq r\} r := r - y; q := 1 + q$
$\{x = r + yq\}$ | consequence(6,9) |
| 11. | $\{x = r + yq\} \mathbf{while} \ y \leq r \ \mathbf{do} \ (r := r - y;$
$q := 1 + q) \{\neg y \leq r \wedge x = r + yq\}$ | while(10) |
| 12. | $\{\mathbf{true}\} \mathbf{PROGRAM} \{desired\ output\}$ | composition(5,11) |

Note how the proof above isn’t *quite* the same result as what we originally derived. The key difference is in the precondition: Our reproduction of Hoare’s proof shows **true** as the precondition, effectively stating “you can start in any program state for the following steps to lead to the postcondition.” In contrast, our first derivation used the precondition $\{x \geq 0 \wedge y > 0\}$, which is slightly more restrictive, since fewer initial program states satisfy this predicate. In fact, the reason we choose the invariant in our first derivation was probably more difficult to understand as a result of this (“Why add the $0 \leq r$?”). Both proofs are valid, so why make one more difficult? Study the program to see how it would behave in case our first precondition is not true when the program is ‘executed.’

²These Lemmas should be proven as an exercise.

1.2. Program Termination

We have shown the “partial correctness” of this introductory example, that is, if the program begins execution in a state satisfying the precondition, and if the program terminates, then the postcondition will be true. But how do we know the program actually terminates? We can often show termination by showing that the following two properties hold:

1. Show that there is some property P which is positive within a loop.
2. Show that for each iteration of the loop, P is decremented by a fixed amount. That is $P_i > P_{i+1}$.

If both properties are true, and if the second property causes P to decrement (yet still be positive), then the only way this can be consistent is for the loop to terminate, or else the first property must become false at some point.

Applying this principle to the previous example: The only variables affecting the loop test are y and r . The former does not change through execution, therefore we concentrate our investigation on what happens to the latter, r . Its initial value is x , which we know from the initialization of the program. In the case that y is strictly greater than x on input, then termination is certain, since the body of the **while** loop would never be executed. Otherwise, r starts out greater than or equal to y and the loop begins execution. In the body of the loop, r is decremented by a positive value (we know it is positive by the precondition); in fact, this decrement is unavoidable. Hence, we may infer that in a finite number of iterations of the loop, r will be decremented to where it is no longer the case that $y \leq r$ and therefore the loop will exit. All execution paths have been accounted for, and hence we conclude that the program will indeed terminate when started in a state satisfying the precondition.

1.3. Example: Multiplication

As a second example we will prove the total correctness of the following program which computes the product of A and B :

$$\begin{array}{l}
\{B \geq 0\} \\
1. \quad MULT(A, B) \equiv \{ \\
2. \quad \quad a := A \\
3. \quad \quad b := B \\
4. \quad \quad y := 0 \\
5. \quad \quad \mathbf{while} \ b > 0 \ \mathbf{do} \\
6. \quad \quad \quad y := y + a \\
7. \quad \quad \quad b := b - 1\} \\
\{y = \overline{AB}\}
\end{array}$$

Note: Since A and B are not modified in the program, we can state the postcondition simply as: $y = AB$.

The general approach is to work backwards, starting with the postcondition, to show that the precondition yields that postcondition. In this case, the program terminates with a **while** statement (lines 5-7). Therefore we need to determine the invariant for the loop. If I is the invariant and X is the **while** statement conditional, then we have to show the following two properties:

1. $\{I \wedge X\} \text{ lines } 6 - 7 \{I\}$
2. $(I \wedge \neg X) \Rightarrow (y = AB)$

The first property will allow us to build the **while** statement, while the second property shows that we get the multiplication answer that we desire.

In order to develop the invariant I , it is necessary to determine what property does not change within the loop. The goal of the loop is to compute AB , which is a constant. Since the loop is increasing y while decreasing b , combining them gives a clue to our invariant. Since ab is just AB initially, we get a clue that our invariant might be something like:

$$y + ab = AB$$

Since we want $b = 0$ at the end, forcing y to be equal to AB , our invariant becomes:

$$I \triangleq (y + ab = AB) \wedge (b \geq 0)$$

Since X is just $b > 0$, $I \wedge \neg X \Rightarrow (y = AB)$ is shown as follows:

$$\begin{aligned}
I \wedge \neg X &\equiv \\
(y + ab = AB) \wedge (b \geq 0) \wedge \neg(b > 0) &\equiv \\
(y + ab = AB) \wedge (b \geq 0) \wedge (b \leq 0) &\equiv \\
(y + ab = AB) \wedge (b = 0) &\equiv \\
(y + a0 = AB) &\equiv \\
(y = AB) &
\end{aligned}$$

In order to show that I is the invariant of the loop:

Line 7. $b := b - 1$

$$\begin{array}{l}
\{y + a(b - 1) = AB \wedge (b - 1) \geq 0\} b := b - 1 \\
\{y + ab = AB \wedge b \geq 0\}
\end{array}
\quad \textit{Assignment axiom}$$

Line 6. $y := y + a$

$$\begin{array}{l}
\{y + a + a(b - 1) = AB \wedge b - 1 \geq 0\} y := y + a \\
\{y + a(b - 1) = AB \wedge b - 1 \geq 0\} \quad \textit{Assign. axiom} \\
y + ab = AB \Rightarrow y + a + a(b - 1) = AB \quad \textit{Theorem} \\
y + ab = AB \Rightarrow (y + a) + a(b - 1) = AB, \\
\{y + a + a(b - 1) = AB \wedge b - 1 \geq 0\} y := y + a \\
\{y + a(b - 1) = AB \wedge b - 1 \geq 0\} \\
\hline
\{y + ab = AB \wedge b - 1 \geq 0\} y := y + a \\
\{y + a(b - 1) = AB \wedge b - 1 \geq 0\} \quad \textit{Consequence}
\end{array}$$

Combining lines 6-7 using Rule of Composition.

$$\frac{\{y + ab = AB \wedge b - 1 \geq 0\} y := y + a \{y + a(b - 1) = AB \wedge b - 1 \geq 0\}, \quad \{y + a(b - 1) = AB \wedge b - 1 \geq 0\} b := b - 1 \{y + ab = AB \wedge b \geq 0\}}{\{y + ab = AB \wedge b - 1 \geq 0\} y := y + a; b := b - 1 \{y + ab = AB \wedge b \geq 0\}}$$

We get the loop invariant as follows:

$$\begin{aligned}
I \wedge X &\equiv \\
(y + ab = AB) \wedge (b \geq 0) \wedge (b > 0) &\Rightarrow \\
(y + ab = AB) \wedge (b > 0) &\Rightarrow \\
(y + ab = AB) \wedge b - 1 \geq 0 &
\end{aligned}$$

By the rule of consequence we get:

$$\{y + ab = AB \wedge b > 0\}y := y + a; b := b - 1\{y + ab = AB \wedge b \geq 0\}$$

The above is just $\{I \wedge X\}$ lines 6 – 7 $\{I\}$. Therefore we can use our **while** axiom:

$$\frac{\{y + ab = AB \wedge b > 0\}y := y + a; b := b - 1\{y + ab = AB \wedge b \geq 0\}}{\{y + ab = AB \wedge b \geq 0\}\mathbf{while} \dots \{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}}$$

We have already shown that this results in $y = AB$.

We finish up by going backwards through the initial assignment showing that the program's precondition yields this invariant:

$$\begin{aligned} \{0 + ab = AB \wedge b \geq 0\}y := 0\{y + ab = AB \wedge b \geq 0\} & \quad \text{Line 4} \\ \{0 + aB = AB \wedge B \geq 0\}b := B\{0 + ab = AB \wedge b \geq 0\} & \quad \text{Line 3} \\ \{0 + AB = AB \wedge (B \geq 0)\}a := A\{0 + aB = AB \wedge B \geq 0\} & \quad \text{Line 2} \end{aligned}$$

Since $B \geq 0 \Rightarrow 0 + AB = AB \wedge B \geq 0$ we get by consequence:

$$\{B \geq 0\}a := A\{0 + aB = AB \wedge B \geq 0\}$$

Combining lines 2-4 by the rule of composition, we complete the proof:

$$\{B \geq 0\}\mathbf{MULT}(\mathbf{A}, \mathbf{B})\{y = AB\}$$

Termination

We have only shown that *if* the program terminates, then it gives us the desired answer. We must also show termination.

1. Let property P be $b > 0$. P is obviously true in loop.
2. The only change to b in loop is $b := b - 1$. So P is decremented in loop.

Since both termination properties are true, the loop must terminate, and we have shown total correctness of the program.

1.4. Another Detailed Example: Fast Exponentiation

Consider the following program to find exponents:

```

{n ≥ 0 ∧ x ≠ 0}
PROGRAM ≡ {
  k := n;
  y := 1;
  z := x;
  while k ≠ 0 do
    if odd(k)
      then k := k - 1; y := y * z
      else k := k/2; z := z * z}
{y = x^n}

```

(As in the previous example, $x = \bar{x}$ and $n = \bar{n}$.) First we identify the invariant. Since k is going down towards zero, we might try something like $y = x^{n-k} \wedge k \geq 0 \wedge x \neq 0$, but this isn't invariant. A better invariant turns out to be $P \equiv yz^k = x^n \wedge k \geq 0 \wedge x \neq 0$.

To establish the invariant, we make three applications of our axiom of assignment:

$$\begin{aligned}
 & \underbrace{\{yx^k = x^n \wedge k \geq 0 \wedge x \neq 0\}}_{\equiv P'} z := x \{P\} \\
 & \underbrace{\{1x^k = x^n \wedge k \geq 0 \wedge x \neq 0\}}_{\equiv P''} y := 1 \{P'\} \\
 & \underbrace{\{1x^n = x^n \wedge n \geq 0 \wedge x \neq 0\}}_{\equiv P'''} k := n \{P''\}
 \end{aligned}$$

We then use these statements in rules of composition:

$$\begin{aligned}
 & \frac{\{P''\}y := 1 \{P'\}, \{P'\}z := x \{P\}}{\{P''\}y := 1; z := x \{P\}} \\
 & \frac{\{P'''\}k := n \{P''\}, \{P''\}y := 1; z := x \{P\}}{\{P'''\}k := n; y := 1; z := x \{P\}}
 \end{aligned}$$

Now, since

$$P''' \equiv (x^n = x^n \wedge n \geq 0 \wedge x \neq 0) \equiv (n \geq 0 \wedge x \neq 0)$$

which is our input condition, then invoking a rule of consequence will establish our invariant.

Next we verify that the loop invariant is invariant. By use of axiom of assignment, we know

$$\underbrace{\{y * (z * z)^k = x^n \wedge k \geq 0 \wedge x \neq 0\}}_{\equiv E'} z := z * z \{P\}$$

and

$$\underbrace{\{y * (z^2)^{k/2} = x^n \wedge k/2 \geq 0 \wedge x \neq 0\}}_{\equiv E''} k := k/2 \{E'\}$$

We can compose these two expressions by

$$\frac{\{E''\}k := k/2 \{E'\}; \{E'\}z := z * z \{P\}}{\{E''\}k := k/2; z := z * z \{P\}}$$

Likewise, by assignment we know

$$\underbrace{\{y * z * z^k = x^n \wedge k \geq 0 \wedge x \neq 0\}}_{\equiv T'} y := y * z \{P\}$$

and

$$\underbrace{\{y * z * z^{k-1} = x^n \wedge k - 1 \geq 0 \wedge x \neq 0\}}_{\equiv T''} k := k - 1 \{T'\}$$

so these in turn can be composed by

$$\frac{\{T''\}k := k - 1 \{T'\}, \{T'\}y := y * z \{P\}}{\{T''\}k := k - 1; y := y * z \{P\}}$$

Next, since

- a: $y * z^k = x^n \wedge x \neq 0 \Rightarrow y * z * z^{k-1} = x^n$
- b: $k \geq 0 \wedge k \neq 0 \Rightarrow k - 1 \geq 0$
- c: $x \neq 0 \Rightarrow x \neq 0$

we can apply the rule of consequence as follows:

$$\frac{P \wedge k \neq 0 \wedge \text{odd}(k) \Rightarrow T'', \{T''\}k := k - 1; y := y * z \{P\}}{\{P \wedge k \neq 0 \wedge \text{odd}(k)\}k := k - 1; y := y * z \{P\}}$$

Likewise, since

$$\begin{aligned} \text{a: } & y * z^k = x^n \wedge x \neq 0 \wedge \neg \text{odd}(k) \Rightarrow y * (z^2)^{k/2} = x^n \\ \text{b: } & k \geq 0 \wedge k \neq 0 \wedge \neg \text{odd}(k) \Rightarrow k/2 \geq 0 \\ \text{c: } & x \neq 0 \Rightarrow x^2 \neq 0 \end{aligned}$$

we can apply a rule of consequence by:

$$\frac{P \wedge k \neq 0 \wedge \neg \text{odd}(k) \Rightarrow E'', \{E''\}k := k/2; z := z * z \{P\}}{\{P \wedge k \neq 0 \wedge \neg \text{odd}(k)\}k := k/2; z := z * z \{P\}}$$

This allows us to apply our **if** rule of inference

$$\frac{\frac{\{P \wedge k \neq 0 \wedge \text{odd}(k)\}k := k - 1; y := y * z \{P\}, \{P \wedge k \neq 0 \wedge \neg \text{odd}(k)\}k := k/2; z := z * z \{P\}}{\{P \wedge k \neq 0\}\text{if } \text{odd}(k) \text{ then } \dots \text{ else } \dots \{P\}}}{\{P \wedge k \neq 0\}\text{if } \text{odd}(k) \text{ then } \dots \text{ else } \dots \{P\}}$$

followed by our **while** rule of inference

$$\frac{\{P \wedge k \neq 0\}\text{if } \text{odd}(k) \text{ then } \dots \text{ else } \dots \{P\}}{\{P\}\text{while } k \neq 0 \text{ do } \dots \{P \wedge k = 0\}}$$

Now we compose the loop with its initialization:

$$\frac{\frac{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x \{P\}, \{P\}\text{while } k \neq 0 \text{ do } \dots \{P \wedge k = 0\}}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x; \text{while } k \neq 0 \text{ do } \dots \{P \wedge k = 0\}}}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x; \text{while } k \neq 0 \text{ do } \dots \{P \wedge k = 0\}}$$

Finally, we show that the postcondition is a consequence of our input:

$$\frac{\frac{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x; \text{while } k \neq 0 \text{ do } \dots \{P \wedge k = 0\}, P \wedge k = 0 \Rightarrow y = x^n}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x; \text{while } k \neq 0 \text{ do } \dots \{y = x^n\}}}{\{n \geq 0 \wedge x \neq 0\}k := n; y := 1; z := x; \text{while } k \neq 0 \text{ do } \dots \{y = x^n\}}$$

1.5. Yet Another Detailed Example: Slow Multiplication

Consider the following (relatively silly) program to perform multiplication:

```

PROGRAM  $\equiv$  {
   $s := 0$ ;
  while  $x \neq 0$  do
     $t := 0$ 
    while  $t \neq y$  do
       $s := s + 1$ ;
       $t := t + 1$ ;
     $x := x - 1$ 
}

```

The desired output condition is $s = \bar{x}\bar{y}$. In this proof, it is convenient to use the following definitions, corresponding to the two loop invariants:

$$\begin{aligned}
 P &\triangleq s = (\bar{x} - x)\bar{y} \wedge y = \bar{y} \\
 Q &\triangleq s = (\bar{x} - x)\bar{y} + t \wedge y = \bar{y}
 \end{aligned}$$

Starting with the innermost **while** statement, we have the following expressions by use of the axiom of assignment:

$$\begin{aligned}
 \{Q_{t+1}^t\} t := t + 1 \{Q\} \\
 \{Q_{t+1}^t s_{s+1}\} s := s + 1 \{Q_{t+1}^t\}
 \end{aligned}$$

By composition of the above lines we infer:

$$\{Q_{t+1}^t s_{s+1}\} s := s + 1; t := t + 1 \{Q\}$$

Next, we observe that

$$\begin{aligned}
 Q_{t+1}^t s_{s+1} &\equiv s + 1 = (\bar{x} - x)\bar{y} + t + 1 \wedge y = \bar{y} \\
 &\equiv s = (\bar{x} - x)\bar{y} + t \wedge y = \bar{y} \\
 &\equiv Q
 \end{aligned}$$

and therefore we trivially know that $Q \wedge t \neq y \Rightarrow Q_{t+1}^t s_{s+1}$, which in turn allows us to apply a rule of consequence to infer

$$\frac{Q \wedge t \neq y \Rightarrow Q_{t+1}^t s_{s+1}, \{Q_{t+1}^t s_{s+1}\} s := s + 1; t := t + 1 \{Q\}}{\{Q \wedge t \neq y\} s := s + 1; t := t + 1 \{Q\}}$$

Our **while** inference rule then allows us

$$\frac{\{Q \wedge t \neq y\} s := s + 1; t := t + 1 \{Q\}}{\{Q\} \mathbf{while} \ t \neq y \ \mathbf{do} \ \dots \{Q \wedge t = y\}}$$

Next, our axiom of assignment provides that

$$\{P_{x-1}^x\} x := x - 1 \{P\}$$

so the rule of consequence implies that

$$\frac{Q \wedge t = y \Rightarrow P_{x-1}^x, \{P_{x-1}^x\} x := x - 1 \{P\}}{\{Q \wedge t = y\} x := x - 1 \{P\}}$$

since

$$\begin{aligned} Q \wedge t = y &\equiv s = (\bar{x} - x) \bar{y} + t \wedge y = \bar{y} \wedge t = y \\ P_{x-1}^x &\equiv s = (\bar{x} - (x - 1)) \bar{y} \wedge y = \bar{y} \\ &\equiv s = (\bar{x} - x) \bar{y} + \bar{y} \wedge y = \bar{y} \end{aligned}$$

and

a: $y = \bar{y} \Rightarrow y = \bar{y}$.

b: $s = (\bar{x} - x) \bar{y} + t \wedge t = y \wedge y = \bar{y} \Rightarrow s = (\bar{x} - x) \bar{y} + \bar{y}$.

Hence we may compose this expression with our **while** loop by:

$$\frac{\{Q\} \mathbf{while} \ t \neq y \ \mathbf{do} \ \dots \{Q \wedge t = y\}, \{Q \wedge t = y\} x := x - 1 \{P\}}{\{Q\} \mathbf{while} \ t \neq y \ \mathbf{do} \ \dots ; x := x - 1 \{P\}}$$

By assignment we know $\{Q_0^t\} t := 0 \{Q\}$, so this can be composed with the previous expression yielding:

$$\{Q_0^t\} t := 0; \mathbf{while} \ t \neq y \ \mathbf{do} \ \dots ; x := x - 1 \{P\}$$

Since $Q_0^t \equiv s = (\bar{x} - x)\bar{y} \wedge y = \bar{y}$ and $P \Rightarrow Q_0^t$, the preceding expression can be used in a rule of consequence to infer

$$\{P \wedge x \neq 0\}t := 0; \text{ while } t \neq y \text{ do } \dots; x := x - 1\{P\}$$

which can then be immediately used in another application of our **while** rule of inference yielding:

$$\{P\}\text{while } x \neq 0 \text{ do } \dots\{P \wedge x = 0\}$$

This expression together with $s = (\bar{x} - x)\bar{y} \wedge y = \bar{y} \wedge x = 0 \Rightarrow s = \bar{x}\bar{y}$ imply

$$\{P\}\text{while } x \neq 0 \text{ do } \dots\{s = \bar{x}\bar{y}\}$$

due to our rule of consequence. Finally, moving into the initialization part of our program, assignment gives us:

$$\{P_0^s\}s := 0\{P\}$$

then composed with the **while** body by

$$\frac{\{P_0^s\}s := 0\{P\}, \{P\}\text{while } x \neq 0 \text{ do } \dots\{s = \bar{x}\bar{y}\}}{\{P_0^s\}s := 0; \text{ while } x \neq 0 \text{ do } \dots\{s = \bar{x}\bar{y}\}}$$

At last, since $P_0^s \equiv 0 = (\bar{x} - \bar{x})\bar{y} \wedge \bar{y} = \bar{y}$, we know by rule of consequence that

$$\frac{\text{true} \Rightarrow P_0^s, \{P_0^s\}s := 0; \text{ while } \dots\{s = \bar{x}\bar{y}\}}{\{\text{true}\} \text{ PROGRAM } \{s = \bar{x}\bar{y}\}}$$

2. CHOOSING INVARIANTS

There are clearly two ‘tough’ problems lurking within this axiomatic approach to proving partial correctness, and these are

- writing a “reasonable” target specification for your program, and
- identifying (i.e. guessing) the invariant for use of our **while** loop inference rule.

For the most part, the first of these above problems is beyond what we have traditionally dealt with in this course, and has been relegated to courses in design and software engineering (if at all). The second of these problems is what stops us from automating the proofs in axiomatic methods.

So the burden of dealing with invariants is shifted to the human problem solver. There are a few points that can partially ease this burden for us. First, we can clearly identify the loop test B from simple inspection. Likewise we know (by the “backwards reasoning” already illustrated in earlier examples here) what assertion will be needed right after execution of the **while** loop. One part of our invariant, then, can be suggested by “factoring out” the $\neg B$ from this assertion. For instance, if by reasoning back we determine that the output condition for the loop must be

$$O \triangleq x \geq 0 \wedge 0 \leq r < y \wedge x = qy + r$$

and if our loop test is $B \equiv r \geq y$, then it follows from our understanding of the inference rule that part of our invariant must be “ $O - \neg B$,” that is,

$$x \geq 0 \wedge 0 \leq r \wedge x = qy + r$$

After coming up with some assertion I , then a check on the proposed invariant that I **and** the negated loop condition are strong enough to prove the output assertion from the loop. If $I \wedge \neg B$ cannot give you the next desired assertion, then there is little reason to spend time insuring that $I \wedge B$ maintain invariance through execution of the loop body.

3. ARRAY ASSIGNMENT

Consider a use of our existing axiom of assignment as in the expression below: