1 Assignmentst 2.5

1.1 Exercises 1

Consider the program C and pre- and postconditions F and H as follows:

- Program C: x = 2;
- Precondition F: $x > 0$
- Postcondition H: $x > 1$

A counterexample for Rule A:
Let G: $x > 2$, using Rule A, if we apply it to the triple ` *ff F gg C ff H gg*, we get:

$$\frac{\text{` } ff \ \mathrm{x} > 0 \ gg \ \mathrm{x} = 2 \ ff \ \mathrm{x} > 1 \ gg}{\text{` } ff \ \mathrm{x} > 0 \ \text{^} \ \mathrm{x} > 2 \ gg \ \mathrm{x} = 2 \ ff \ \mathrm{x} > 1 \ \text{^} \ \mathrm{x} > 2 \ gg} \ \mathrm{ruleA}$$

We have: j= *ff* x > 0 *gg* x = 2 *ff* x > 1 *gg*, which is true. However, simplifying the conditions, we have ` *ff* x > 2 *gg* x = 2 *ff* x > 2 *gg*, the postcondition $x > 2$ is not satisfed by the program, since after execution, we have $x = 2$ which doesn't satisfy the condition.

A Counterexample for Rule B:
Let a = x+1, using Rule B, if we apply it to the triple ` *ff F gg C ff H gg*, we get:

$$\frac{\text{` } ff \ \mathrm{x} > 0 \ gg \ \mathrm{x} = 2 \ ff \ \mathrm{x} > 1 \ gg}{\text{` } ff \ \mathrm{x} > 1 \ gg \ \mathrm{x} = 2 \ ff \ \mathrm{x} > 2 \ gg} \ \mathrm{ruleA}$$

We have: j= *ff* x > 0 *gg* x = 2 *ff* x > 1 *gg*, which is true. However, simplifying the conditions, we have ` *ff* x > 1 *gg* x = 2 *ff* x > 2 *gg*, the postcondition $x > 2$ is not satisfed by the program, since after execution, we have $x = 2$ which doesn't satisfy the condition.

In both cases, we have shown that applying the proposed rules can lead to incorrect conclusions, which demonstrates their unsoundness.

1.2 Exercises 2

To show that we can verify all triples *ff F gg C ff H gg* if we know how to verify triples of the form *ff* `true` *gg* C^0 *ff* `true` *gg*, we'll construct a command C^0.

Let's consider the verifcation conditions:

1. For *ff* `true` *gg* C^0 *ff* `true` *gg*, we have `true` ! wp[C^0](`true`), which simplifes to `true` ! `true`, which is trivially true.
2. For *ff F gg C ff H gg*, we have F ! wp[C](H).

Now, we want to construct C^0 such that:

$$\text{` } ff \ \mathtt{true} \ gg \ C^0 \ ff \ \mathtt{true} \ gg \quad ( \ ) \quad \text{` } ff \ F \ gg \ C \ ff \ H \ gg$$

The idea is to make C^0 a sequential composition of two commands:

$$C^0 = \mathtt{assume}\ F; C; \mathtt{assert}\ H$$

Now, let's justify why this construction is correct:

1. $\Rightarrow$ (Forward direction)

   Assume $\vdash \{\texttt{true}\}\ C'\ \{\texttt{true}\}$. This means $\texttt{true} \rightarrow \mathrm{wp}[C'](\texttt{true})$.

   By definition of weakest precondition, $\mathrm{wp}[C'](\texttt{true})$ is the weakest formula that holds before the execution of C' and guarantees that `true` holds after the execution of C'.

   Since `true` always holds, this means that $\mathrm{wp}[C'](\texttt{true})$ must also always hold.

   Therefore, $F \rightarrow \mathrm{wp}[C](H)$, as required.

2. $\Leftarrow$ (Backward direction)

   Assume $\vdash \{F\}\ C\ \{H\}$. This means $F \rightarrow \mathrm{wp}[C](H)$.

   We want to show that $\vdash \{\texttt{true}\}\ C'\ \{\texttt{true}\}$, i.e., $\texttt{true} \rightarrow \mathrm{wp}[C'](\texttt{true})$.

   By definition of weakest precondition, $\mathrm{wp}[C'](\texttt{true})$ is the weakest formula that holds before the execution of C' and guarantees that `true` holds after the execution of C'.

   The command C' is constructed such that it first assumes F, then executes C, and finally asserts H.

   This means that the weakest formula that holds before the execution of C' is F. And since `true` always holds, the condition `true` holds after the execution of C'.

   Therefore, $\texttt{true} \rightarrow \mathrm{wp}[C'](\texttt{true})$, as required.

In conclusion, by constructing C' as $\texttt{assume}\ F;C;\texttt{assert}\ H$, we have shown that we can verify all triples $\{F\}\ C\ \{H\}$ if we know how to verify triples of the form $\{\texttt{true}\}\ C'\ \{\texttt{true}\}$.

1.3 Exercises 3

```
method Example1() {
  var x: Int
  var y: Int

  assume x == y
  x := y + 1
  assert x > y
}
```

In this example, the program assumes that x and y are equal, assigns x the value y + 1, and asserts that x is greater than y. The program is functional, partial and total correct.

```
method Example2() {
  var x: Int
  assert x > 0
}
```

In this example, the program asserts that x is greater than 0. The program is functional correct, but not partial and total correct.

```
method Example3() {
  var x: Int

  while (true) {
```

```
    x := 1
  }
  assert x == 0
}
```

In this example, the program starts with an while loop, fnally asserts that x equals to 0. The program is functional and partial correct, but not total correct.

Example 1 and 3 pass Viper verifcation, while the second one failed, which means Viper verifes programs with respect to partial correct.

1.4 Exercises 4

Now, we can defne the safe transformer as follows if our command contains assert F:

safe$[C](F)$ = sp$[C](safe[C'](F))$ where C' is the remaining portion of the command if $C = C1;C2;C3...$

The key idea is to recursively apply the strongest postcondition transformer to calculate the logical formula safe$[C](F)$ that represents the safety condition for running command C with precondition F.