Exercise 4F-2. VCGen for Let [6 points].

The given set of rules for the VC is:

VC(c1; c2, B) = VC(c1, VC(c2, B))

VC(x := e, B) = [e/x] B

VC(let x = e in c, B) = [e/x] VC(c, B)

But these rules turn out to have a bug. [VC(let x = e in c, B) = [e/x] VC(c, B)]

The issue arises because x might be assigned a new value within c, meaning that blindly substituting e for x in VC(c, B) could lead to incorrect reasoning.

Thus, to properly handle the scope of x, we make sure that the VC is computed within the correct variable context.

Thus, the corrected form of the rule is:

VC(let x=e in c,B)=VC(c,[e/x]B)

Here, instead of substituting e for x in VC(c, B), we apply [e/x] only to B, ensuring that x's definition is correctly propagated forward while keeping VC(c, B) intact in its proper scope.

Hence, this prevents unintended substitutions within VC(c, B), particularly when c reassigns x to another value.

Exercise 4F-3. VCGen Mistakes [6 points]

Here, to demonstrate the unsoundness of the buggy let rule, we construct an example as follows:

Step 1: Choosing a command c

C = let x=0 in x:=1, here we take a simple let statement where x is assigned to 0 and then x Is reassigned within the block.

Step 2: Choosing a Post-Condition B

B: x = 0

Here we are expecting that, even after execution x can be still 0.

Step 3: Choosing a State σ

Let us assume the initial state σ where x is undefined. (No value assigned yet)

Step 4: Checking the Buggy VC Rule

The discussed buggy VC rule in class is: VC(let x=e in c,B)=[e/x]VC(c,B)

Questions assigned to the following page: 3 and 4

Here, substituting e = 0, c = (x := 1), and B = (x = 0), we get:

VC(let x=0 in x:=1,x=0)=[0/x]VC(x:=1,x=0)

But when we apply the rule for assignment,

VC(x:=1,x=0)=[1/x](x=0)

=(1=0), which is FALSE

Since substituting 0 for x before generating VC(c, B) leads to (1 = 0), which is trivially false, the verification condition always holds regardless of σ. This means:

$\sigma \models VC(c,B)$ (since VC is always false, it's trivially true in logic)

Step 5: Executing c

Now executing c in the actual program,

1. x = 0 is initialized
2. x:=1 is executed, so x is now 1

Thus, after execution, the new state is σ' where the x = 1

Hence, <c, σ> ⇓ σ'

Step 6: Checking σ' |= B

We know that the post condition B was x=0, but after the execution, x = 1.

Thus clearly,

σ' not |= B (I am unable to copy paste the exact symbol for not from the HW4 pdf)

which shows that the verification condition is incorrectly satisfied, but the program does not actually ensure the post-condition.

On the whole, we can say that the let rule allows the verification to pass, but in reality the program does not satisfy the post condition. Thus this proves that the buggy rule is unsound.

Exercise 4F-4. Axiomatic Do-While [6 points].

So as we know, the semantics for do c while b are as follows

1. Execute c at least once
2. Check condition b
3. If b is true, repeat c, else exit

The Hoare Triple can be written as

{P} c {Q}

Question assigned to the following page: 4

P refers to the pre condition and Q refers to the post condition.

Thus, this is created such that:
1. P holds before entering the loop
2. C executes at least once

3. If b is true, c repeats

4. If b is false, execution exits with Q holding

Now, we introduce the loop variant I that should hold before the first execution of c and after each iteration of c.

Thus the hoare rule for do c while b is:

$$\frac{\{I\}\ c\ \{I\} \quad \{I \wedge \neg b\} \Rightarrow Q}{\{I\}\ \text{do } c \text{ while } b\ \{Q\}}$$

Here, c preserves I.

Exit condition: $I \wedge \neg b \rightarrow Q$
When b is false, the execution stops.
And then the post condition Q must follow from $I \wedge \neg b$

Now, let us consider an example and make sure this rule is correct.

```
do {
  x := x - 1;
} while (x > 0);
```

We want to prove $\{ x \geq 0 \}$ as the precondition and $\{ x = 0 \}$ as the postcondition.

1. Loop Invariant (I): $x \geq 0$
Before the first iteration, x is non-negative. After each iteration, x decreases but remains non-negative.

2. Execution of c: x := x - 1
If $x > 0$, it remains valid that $x \geq 0$.

3.Exit Condition ($I \wedge \neg b \rightarrow Q$):
The loop stops when $x \leq 0$, so $Q = (x = 0)$ holds.

Thus we can see that the rule we created is complete and sound.