Exercise 4F-2. VCGen for Let

The given verification condition generation (VCGen) rule for `let` is:

$$VC(\text{let } x = e \text{ in } c, B) = [e/x]VC(c, B)$$

This rule is incorrect because it applies the substitution $[e/x]$ too early, potentially leading to incorrect variable scoping.

Corrected Rule

The correct rule should be:

$$VC(\text{let } x = e \text{ in } c, B) = VC(c, [e/x]B)$$

- The original rule incorrectly substitutes e for x in the entire verification condition of c, which may lead to unintended variable capture.
- The correct approach first computes $VC(c, B)$, ensuring that the verification condition for c is properly derived.
- Only after computing $VC(c, B)$, we apply the substitution $[e/x]$ to the resulting postcondition B.
- This ensures that the substitution is correctly scoped and only affects the part of the verification condition where x is in scope.

Thus, the corrected rule preserves the correct handling of variable scope in backward verification condition generation.

Exercise 4F-3. VCGen Mistakes

We need to demonstrate the unsoundness of the buggy verification condition (VC) generation rule for `let` expressions, formulated as:

$$\text{VC}(\text{let } x = e \text{ in } c \text{ end}, B) = \text{VC}(c, B)[e/x]$$

where $[e/x]$ represents substituting all occurrences of x in the verification condition with the expression e.

1. Command c

```
y := 0;
let y = 5 in
  skip
end
```

This command:

- Sets $y := 0$.
- Introduces a local binding $y = 5$ within the `let` block.
- Executes `skip`, which does nothing.
- Exits the `let` block, removing the local $y = 5$ binding.

2. Post-condition B

$$B : y = 5$$

We expect y to be 5 after execution.

3. Initial State σ

$$\sigma = \{\}$$

An empty state where y is initially undefined.

4. Verification Condition $\sigma \models VC(c, B)$

Using the buggy rule:

$$\text{VC}(y := 0; \text{let } y = 5 \text{ in skip end}, y = 5)$$

Expanding step-by-step:

$$
\begin{aligned}
&\text{VC}(y := 0, \text{VC}(\text{let } y = 5 \text{ in skip end}, y = 5))\\
&\qquad \text{VC}(\text{let } y = 5 \text{ in skip end}, y = 5) = \text{VC}(\text{skip}, y = 5)[5/y] \quad \text{(Applying buggy rule)}\\
&\qquad = (y = 5)[5/y] \quad \text{(Since VC of } \texttt{skip} \text{ is just the post-condition)}\\
&\qquad = 5 = 5 \quad \text{(Substituting } y \text{ with 5, which is always true)}\\
&\qquad = \text{true}
\end{aligned}
$$

Now applying this to the outer assignment:

$$\text{VC}(y := 0, \text{true}) = \text{true}$$

Since the verification condition holds unconditionally, we conclude:

$$\sigma \models VC(c, B)$$

5. Execution of $\langle c, \sigma \rangle$

- $y := 0 \Rightarrow \sigma = \{y \mapsto 0\}$
- `let` $y = 5$ in skip end
  - Creates a local binding $y = 5$.
  - Executes `skip`, which does nothing.
  - Exits the block, discarding the local $y = 5$.
- Final state: $\sigma' = \{y \mapsto 0\}$.

6. Post-condition Failure $\sigma' \not\models B$

Since $B : y = 5$ and the final state has $y = 0$, we conclude:

$$\sigma' \not\models B$$

The verification condition was satisfied initially, but execution led to a final state that **did not satisfy** the post-condition. This demonstrates that the buggy `let` rule is **unsound** because it incorrectly substitutes variables without respecting scope. A correct rule must properly handle variable shadowing to ensure sound verification.

Exercise 4F-4. Axiomatic Do-While

To derive a sound and complete Hoare logic rule for the `do c while b` loop, we need to capture its semantics:

- The body c is executed **at least once** before b is tested.
- The loop continues executing as long as b holds.
- When the loop terminates, b must be false.

Hoare Rule

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

Explanation of the Rule

1. **First execution:** The loop executes at least once, so the precondition P must ensure that executing c at least once establishes the loop invariant I:
$$\{P\}\ c\ \{I\}$$
2. **Loop invariant preservation:** The invariant I must hold before and after each execution of c, as long as b is true:
$$\{I \wedge b\}\ c\ \{I\}$$
3. **Termination condition:** When the loop terminates, b must be false. The final post-condition Q must be implied by the invariant and the negation of b:
$$(I \wedge \neg b) \Rightarrow Q$$

Soundness and Completeness

- **Soundness:** If P holds before execution, and the invariant I is maintained correctly, then Q will hold when the loop terminates.
- **Completeness:** This rule allows us to prove correctness for any valid `do c while b` loop by selecting an appropriate loop invariant I.

Example

Consider the program:

```
do
  x := x + 1
while x < 10
```

with:

- **Precondition:** $P : x = 0$
- **Postcondition:** $Q : x = 10$
- **Loop invariant:** $I : x \leq 10$

Applying the Rule

1. **First execution establishes the invariant:**

$$\{x = 0\}\ x := x + 1\ \{x \leq 10\}$$

   After execution, $x = 1$, which satisfies $x \leq 10$.

2. **Loop invariant preservation:**

$$\{x \leq 10 \wedge x < 10\}\ x := x + 1\ \{x \leq 10\}$$

   If $x < 10$, then incrementing x ensures $x \leq 10$ still holds.

3. **Termination ensures $x = 10$:**

$$(x \leq 10 \wedge \neg(x < 10)) \Rightarrow x = 10$$

   Since $\neg(x < 10)$ implies $x \geq 10$, and we already have $x \leq 10$, it follows that $x = 10$.

Thus, we prove:

$$\{x = 0\}\ \text{do}\ x := x + 1\ \text{while}\ x < 10\ \{x = 10\}$$

The Hoare rule for the `do c while b` construct ensures that the loop executes at least once, maintains the loop invariant, and satisfies the postcondition upon termination. This rule is both sound and complete.