2 Exercise 4F-2. VCGen for Let

The difference between let and assignment is that the value returns to its previous value upon the completion of the scope.

The given rule works if used when the let is the only scope, but will fail when combined with other rules, as it will assume the value is modified still.

We can derive the correct rule by desugaring let $x = e$ in c into its corresponding statements:

$$\begin{aligned} x &= e; \\ &c; \\ x &= x_{old}; \end{aligned}$$

Where x_{old} is the value of x prior to the let statement.

Running VCGen from bottom to top on these statements, we get the modified rule:

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

Which now reflects that the condition after must still hold after returning x to its previous value.

3 Exercise 4F-3.VCGen Mistakes

We can give the following IMP code to demonstrate the unsoundness of the let rule:

$$\begin{aligned} &x = 0; \\ &\text{let } x = 10 \text{ in } skip \end{aligned}$$

With the post condition $B = x > 1$.

Running VCGen, we end up with the condition of $x > 1$ after the let body, and then $10 > 1 \rightarrow$ True from above the let. True is the weakest precondition, and thus we infer that no other preconditions are needed for A, even after the assignment.

However, if we run this code, x returns to 0 after the let expression, and thus violates the post condition. Therefore, the original let VCGen rule is unsound.

4 Exercise 4F-4. Do-While

Similar to before, we can desugar the do-while statement into other primitives, and then run VCGen on them. do c while b is equivalent to:

$$\begin{array}{l} c; \\ \ \text{while } b \text{ do } c \end{array}$$

Running VCGen on this, we can combine the while rule with the semicolon rule, getting:

$$VC(\text{ do } c \text{ while } b) = VC(c, b \to VC(c, b) \wedge \neg b \to B)$$