1 4F-1 Bookkeeping

- 0 pts Correct

Exercise 4F-2. VCGen for Let [6 points]. In class we gave the following rules for the (backward) verification condition generation of assignment and let:

$$\begin{array}{ll} \text{VC}(c_1; c_2, B) & = \text{VC}(c_1, \text{VC}(c_2, B)) \\ \text{VC}(x := e, B) & = [e/x]\ B \\ \text{VC}(\mathsf{let}\ x = e\ \mathsf{in}\ c, B) & = [e/x]\ \text{VC}(c, B) \end{array}$$

That rule for let has a bug. Give a correct rule for let.

- The main mistake is the fact that, although it binds x to a value as with the assignment, this binding does not stay local to the let statement.
- One way to enforce the scope of the substitution is to prevent it from modifying anything else outside the original let statement. i.e. it "leaks" out into the main program's scope and can erroneously satisfy some
- The basic idea is that some post-condition constraints may contain variables with the same name as the variable x in the let statement. Therefore, we need to "shield" these from the substitution.
- Find some label x' that is unbound (fresh) in the entire program, and temporarily replace all occurrences of x with x' in the post-condition: $\exists x'.(\mathsf{fresh}\ x')\ s.t.\ [x'/x]B$. This is always possible since there are infinite strings to choose from, yet programs (and variable usages) are finite.
- Perform the original substitution $[e/x]$, which should replace only those x which occur strictly inside the scope of the let statement.
- There should be no x anywhere in the resulting VC due to the previous replacement.
- Finally, replace all previously-guarded variables x' with their original x. If there are no occurrences of x in the previous program, this is trivially satisfied.

All together, the result is

$$\text{VC}(\mathsf{let}\ x = e\ \mathsf{in}\ c, B) = \boxed{[x/x']([e/x]\text{VC}(c, [x'/x]B)), \exists x'.(\mathsf{fresh}\ x')}$$

As mentioned above, assume we can find a fresh variable x' (doesn't have to be literally called x') that we can use temporarily as a substitution.

As an aside (unrelated to the above), I considered another approach: $\text{VC}(c, B)\backslash_1\{x = e\}$ where $\backslash_1\{x = e\}$ removes one algebraically-equivalent constraint from B that is satisfied by $x = e$. The reasoning was that any constraints involving $x = e$ that were created inside the let statement could simply be disregarded, since they will be bound upon entry into the let. However, this rule assumes access to an algebraic solver to help make such decisions, and may not find constraints that differ from the $x = e$ pattern, such as $x \leq e$. In addition, it does not account for the possibility of multiple constraints involving $x = e$ generated inside the let statement. As a result, this rule might not be complete.

2 4F-2 VCGen for Let

- 0 pts Correct

Exercise 4F-3. VCGen Mistakes [6 points]. Given $\{A\}c\{B\}$ we desire that $A \implies \text{VC}(c, B) \implies \text{WP}(c, B)$. We say that our VC rules are *sound* if $\models \{\text{VC}(c, B)\}\ c\ \{B\}$. Demonstrate the unsoundness of the buggy `let` rule by giving the following six things:

1. a command c: `let x := 5 in skip`
2. a post-condition B: {`x=5`}
3. a state σ: {`x:=0`}
4. such that $\sigma \models \text{VC}(c, B)$:
   - $\text{VC}(c, B) \equiv \text{VC}(\texttt{let x := 5 in skip}, \{x = 5\})$
   - $\equiv [5/x]\text{VC}(\texttt{skip}, \{x = 5\})$
   - $\equiv [5/x]\{x = 5\}$
   - $\equiv \{5 = 5\}$
   - $\sigma \models \{5 = 5\}$
5. and $\langle c, \sigma\rangle \Downarrow \sigma'$:
   - $\langle \texttt{let x := 5 in skip}, \sigma\rangle \Downarrow \sigma' = \sigma = \{x := 0\}$
6. but $\sigma' \not\models B$:
   - $\sigma' \not\models \{x = 5\}$ since $x := 0$ and $0 \neq 5$

3 4F-3 VCGen Mistakes

- 0 pts Correct

Exercise 4F-4. Axiomatic Do-While [6 points]. Write a sound and complete Hoare rule for $\textsf{do}\ c\ \textsf{while}\ b$. This statement has the standard semantics (e.g., c is executed at least once, before b is tested).

$\textsf{do}\ c\ \textsf{while}\ b$ can be reduced to a sequence of two existing commands: c; $\textsf{while}\ b\ \textsf{do}\ c$

Therefore, we can adapt the existing rules for these two commands.

Suppose that, given pre-condition $\{A\}$, executing c results in $\{B\}$: $\{A\}c\{B\}$

We know that c is guaranteed to be executed at least once (in the case where $b = \textsf{false}$), so we know the end state should be at least $\{B\}$, plus whatever we can say about b at the end: $\{B \wedge \neg b\}$

It would also be nice to have the same properties hold no matter how many additional times we go around the loop (even though zero seems like a good number of times to run around a loop). Therefore, let's also ensure that satisfying the loop guard b preserves the post-condition $\{B\}$: $\{B \wedge b\}cB$

Therefore, we end up with the following Hoare rule for $\textsf{do}\ c\ \textsf{while}\ b$:

$$\frac{\vdash \{A\}c\{B\} \quad \vdash \{B \wedge b\}c\{B\}}{\vdash \{A\}\ \textsf{do}\ c\ \textsf{while}\ b\ \{B \wedge \neg b\}}$$

4 4F-4 Axiomatic Do-While

- 0 pts Correct