# Axiomatic Semantics of Loops

## 1 VC For Let

First, we unwind `let`, where `t` is a fresh variable:

$$\text{let x = e in c} \quad \equiv \quad \text{t := x; x := e; c; x := t}$$

Thus we compute the VC:

$$\text{VC(let x = e in C}, B) \quad \equiv \quad \text{VC(t := x; x := e; c; x:= t}, B)$$

So the result is:

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

An alternative formulation is to apply substitution to the command:

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

## 2 Unsound Let

The basic problem with the buggy let rule is that it does not restore the old value.

1. $c = \text{let x = 1 in skip}$
2. $B = \text{x} = 1$
3. $\sigma(x) = 0$
4. $\sigma \models VC(c, b)$, since $VC(c, b)$ is $1 = 1$
5. $\langle c, \sigma \rangle \Downarrow \sigma'$, where $\sigma'(x) = 0$, because the original value is restored after a let
6. $\sigma' \not\models B$ because $\sigma' \not\models$ x=1 because $\sigma'(x) = 0$.

## 3 Hoare Rule

First, we unwind `do-while`:

$$\text{do c while b} \quad \equiv \quad \text{c ; while b do c}$$

We will obtain our final Hoare rule by substituting in the appropriate rules:

$$\frac{\vdash \{A\}\text{c}\{B\} \quad \vdash \{B\}\text{while b do c}\{C\}}{\vdash \{A\}\text{do c while b}\{C\}}$$

That answer is officially good enough. You can view it as using the "alternate" `while` rule given on the *Alternate Hoare Rules* slide (near page 16) of the *Introduction To Axiomatic Semantics* lecture. We could also rephrase it using the more common `while` rule:

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

Which is then equivalent to:

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

# 4 Backwards VC

First, we unwind `do-while`:

$$\text{do}_{Inv} \text{ c while b} \quad \equiv \quad \text{c ; while}_{Inv} \text{ b do c}$$

Now we can compute the VC:

$$\begin{aligned}
&\text{VC(c; while}_{Inv} \text{ b do c}, B) && = \\
&\text{VC(c}, \text{VC(while}_{Inv} \text{ b do c}, B)) && = \\
&\text{VC(c}, Inv \wedge (\forall x_1 \ldots x_n.\ Inv \implies ((b \implies VC(c, Inv)) \wedge \neg b \implies B))) && = \\
&\text{VC(c}, Inv) \wedge (\forall x_1 \ldots x_n.\ Inv \implies ((b \implies VC(c, Inv)) \wedge \neg b \implies B))
\end{aligned}$$

where $x_1 \ldots x_n$ are the variables modified in $c$.

For the alternate formulation (4B), we again start by unwinding:

$$\text{do}_{Inv1,Inv2} \text{ c while b} \quad \equiv \quad \text{assert}(Inv1)\text{; c ; while}_{Inv2} \text{ b do c}$$

The assertion comes from the problem description that $Inv1$ is true before and after $c$ is executed. We use $Inv2$ to refer to the loop invariant of the while loop; it is typically the same as $Inv1$, but may potentially be stronger (i.e., it may incorporate information from the first execution of c). Note that $Inv2$ must imply $Inv1$ since $Inv1$ must also be true on every iteration of the while loop. So the result is:

$$\begin{aligned}
&\text{VC(assert}(Inv1 \wedge Inv2 \Rightarrow Inv1)\text{; c; while}_{Inv2} \text{ b do c}, B) && = \\
&\text{VC(assert}(Inv1 \wedge Inv2 \Rightarrow Inv1), \text{VC(c; while}_{Inv2} \text{ b do c}, B)) && = \\
&Inv1 \wedge Inv2 \Rightarrow Inv1 \wedge \text{VC(c; while}_{Inv2} \text{ b do c}, B) && = \\
&Inv1 \wedge Inv2 \Rightarrow Inv1 \wedge \text{VC(c}, \text{VC(while}_{Inv2} \text{ b do c}, B)) && = \\
&Inv1 \wedge Inv2 \Rightarrow Inv1 \wedge \text{VC(c}, Inv2 \wedge (\forall x_1 \ldots x_n.\ Inv2 \implies (b \implies VC(c, Inv2)) \wedge \neg b \implies B))
\end{aligned}$$

where $x_1 \ldots x_n$ are the variables modified in $c$.

## 4.1 Common Mistake

The fact that the VC encodes the first execution of the command $c$ is critical. One common mistake was to use something like the while rule from class:

$$Inv \wedge (\forall x_1 \ldots x_n.\ Inv \implies (b \implies VC(c, Inv)) \wedge \neg b \implies B))$$

Consider the program "do x := 1 while false". The program is basically an assignment statement dressed up as a loop. We should be able to compute the VC of it with respect to the post-condition $x = 1$. We expect that VC to be equivalent to "true". Unfortunately, with the mistaken rule, we get:

$$\begin{aligned}
&Inv \wedge (\forall x.\ Inv \implies (false \implies VC(x := 1, Inv)) \wedge true \implies x = 1)) && = \\
&Inv \wedge (\forall x.\ Inv \implies true \implies x = 1)) && = \\
&Inv \wedge (\forall x.\ Inv \implies x = 1))
\end{aligned}$$

There is no value of $Inv$ for which this works. If we take $Inv$ to be $x = 1$ we satisfy the right conjunct but cannot sastify the left. If we take $Inv$ to be true, we satisfy the left but cannot satisfy the right.

If we do not use the mistaken rule but instead use the correct rule above, we get the following VC:

$$\begin{aligned}
&\underline{\text{VC(x:=1}, x = 1)} \wedge (\forall x.\ Inv \implies (false \implies \text{VC(x:=1}, Inv)) \wedge (true \implies x = 1)) && = \\
&\underline{\text{VC(x:=1}, x = 1)} \wedge (\forall x.\ Inv \implies (true \implies x = 1)) && =
\end{aligned}$$

We can take $Inv$ to be $x = 1$:

$$\begin{aligned}
\underline{\text{VC(x:=1}, x = 1)} \wedge (\forall x.\ x = 1 \implies (true \implies x = 1)) &= \\
\underline{\text{VC(x:=1}, x = 1)} \wedge (\forall x.\ x = 1 \implies x = 1) &= \\
\underline{\text{VC(x:=1}, x = 1)} \wedge (\forall x.\ true) &= \\
\underline{\text{VC(x:=1}, x = 1)} &= \\
\underline{[1/x]}x = 1 &= \\
1 = 1 &= \\
true &
\end{aligned}$$