

Program Verification

Exercise Solutions 6: Hoare Logic and Weakest Preconditions

Assignment 1 (Alternative Hoare Logic Rules)

1. We consider the question for each statement in turn; for each, we need to show that, given an arbitrary instance of the alternative rule, the same triple could be derived without the alternative rule:

havoc: Then, given the form of the havoc-alt rule, it suffices to show (for arbitrary A , x) that the Hoare triple $\{A\} \text{havoc } x \{\exists y. A[y/x]\}$ is derivable using the usual rules. One possible derivation is as follows:

$$\frac{\frac{\frac{}{\{\forall z. (\exists y. A[y/x])[z/x]\} \text{havoc } x \{\exists y. A[y/x]\}} \text{(havoc)}}{\{\exists y. A[y/x]\} \text{havoc } x \{\exists y. A[y/x]\}} \text{(conseq)}}{\{A\} \text{havoc } x \{\exists y. A[y/x]\}} \text{(conseq)}$$

Note that the upper of the two rule-of-consequence steps is justified by the fact that x does not occur in $\exists y. A[y/x]$, and so $\forall z. (\exists y. A[y/x])[z/x]$ is the same as $\forall z. (\exists y. A[y/x])$ in which z does not occur under the quantifier, which is equivalent to $\exists y. A[y/x]$.

assert: Then, given the form of the assert-alt rule, it suffices to show (for arbitrary A , A_1) that the Hoare triple $\{A\} \text{assert } A_1 \{A_1 \wedge A\}$ is derivable *assuming* that $A \models A_1$ (since this premise is a requirement of the assert-alt rule). We can show this as follows:

$$\frac{\frac{}{\{A_1 \wedge A_1 \wedge A\} \text{assert } A_1 \{A_1 \wedge A\}} \text{(assert)}}{\{A\} \text{assert } A_1 \{A_1 \wedge A\}} \text{(conseq)}$$

Note that the rule-of-consequence step is justified because of the assumption that $A \models A_1$.

assume: Then, given the form of the assume-alt rule, it suffices to show (for arbitrary A , A_1) that the Hoare triple $\{A\} \text{assume } A_1 \{A_1 \wedge A\}$ is derivable. We can show this as

follows:

$$\frac{\frac{}{\{A_1 \Rightarrow A \wedge A_1\} \text{ assume } A_1 \{A_1 \wedge A\}} \text{ (assume)}}{\{A\} \text{ assume } A_1 \{A_1 \wedge A\}} \text{ (conseq)}$$

Note that the rule-of-consequence step is justified, since $A \models (A_1 \Rightarrow A \wedge A_1)$.

2. Similarly, we show that an arbitrary instance of the normal rule can be derived using the analogous alternative rule (plus the other normal ones).

havoc: Then, given the form of the havoc rule, it suffices to show (for arbitrary A, x) that the Hoare triple $\{\forall y. A[y/x]\} \text{ havoc } x \{A\}$ is derivable using the alternative (havoc-alt) rule, and the other normal rules. One possible derivation is as follows:

$$\frac{\frac{\frac{}{\{\forall y. A[y/x]\} \text{ havoc } x \{\exists z. (\forall y. A[y/x])[z/x]\}} \text{ (havoc-alt)}}{\{\forall y. A[y/x]\} \text{ havoc } x \{\forall y. A[y/x]\}} \text{ (conseq)}}{\{\forall y. A[y/x]\} \text{ havoc } x \{A\}} \text{ (conseq)}$$

Note that the upper of the two rule-of-consequence steps is justified by the fact that x does not occur in $\forall y. A[y/x]$, similarly to in the first part of the question.

assert: Then, given the form of the assert rule, it suffices to show (for arbitrary A, A_1) that the Hoare triple $\{A_1 \wedge A\} \text{ assert } A_1 \{A\}$ is derivable. We can show this as follows:

$$\frac{\frac{}{\{A_1 \wedge A\} \text{ assert } A_1 \{A_1 \wedge A_1 \wedge A\}} \text{ (assert-alt)}}{\{A_1 \wedge A\} \text{ assert } A_1 \{A\}} \text{ (conseq)}$$

Note that the assert-alt step is justified because $A_1 \wedge A \models A_1 \wedge A_1 \wedge A$.

assume: Then, given the form of the assume rule, it suffices to show (for arbitrary A, A_1) that the Hoare triple $\{A_1 \Rightarrow A\} \text{ assume } A_1 \{A\}$ is derivable. We can show this as follows:

$$\frac{\frac{}{\{A_1 \Rightarrow A\} \text{ assume } A_1 \{(A_1 \Rightarrow A) \wedge A_1\}} \text{ (assume-alt)}}{\{A_1 \Rightarrow A\} \text{ assume } A_1 \{A\}} \text{ (conseq)}$$

Assignment 2 (Desugaring If-Conditions)

For arbitrary b, s_1, s_2, A , we have:

$$\begin{aligned} wlp(\text{if}(b)\{s_1\}\text{else}\{s_2\}, A) &= (b \Rightarrow wlp(s_1, A)) \wedge (\neg b \Rightarrow wlp(s_2, A)) \\ &= wlp(\text{assume } b, wlp(s_1, A)) \wedge wlp(\text{assume } \neg b, wlp(s_2, A)) \\ &= wlp(\text{assume } b; s_1, A) \wedge wlp(\text{assume } \neg b; s_2, A) \\ &= wlp((\text{assume } b; s_1)[](\text{assume } \neg b; s_2), A) \end{aligned}$$

Assignment 3 (Dynamic Single Assignment)

A possible optimisation of the treatment of branching constructs is to avoid introducing an additional version of a variable at the end of two branches, if the most-recent version in each branch was the same. Better yet, if we assume we have some scheme of indexing variable versions by an integer, and we number the versions of variables used consecutively (say, original program variable x has versions $x_{(0)}, x_{(1)}, x_{(2)}, \dots$), then we can add an additional assignment only to the branch which had a lower version number. For example, the following program (a slight variant of that in the question):

$x:=3; \text{if } (y > 4) \{x:=x+1\} \text{ else } \{x:=x+1; x:=x+1\}; \text{assert } x > 1$

could be converted to

$x_0:=3; \text{if } (y_0 > 4) \{x_1:=x_0+1; x_2:=x_1\} \text{ else } \{x_1:=x_0+1; x_2:=x_1+1\}; \text{assert } x_2 > 1$

Combining this with the other requirements from the question, we define our operator *toDSA* in terms of the input/output statements, and a (mathematical) map \mathcal{V} from (the original) program variables to integers. We write $\mathcal{V}[x]$ for map lookup, $\mathcal{V}[x \mapsto i]$ for map update (defined as usual; this defines a new map in terms of the old map \mathcal{V} , but doesn't change the definition of \mathcal{V} itself), and (useful in our definition below), we define a *maximum* operator of maps: $\max(\mathcal{V}_1, \mathcal{V}_2)$ is a

map defined pointwise by: $\max(\mathcal{V}_1, \mathcal{V}_2)[i] = \begin{cases} \mathcal{V}_1[i] & \text{if } \mathcal{V}_1[i] \geq \mathcal{V}_2[i] \\ \mathcal{V}_2[i] & \text{otherwise} \end{cases}$

We pass pairs of a statement and map \mathcal{V} as input and result from our *toDSA* operation; the map keeps track of the latest version of the program variable; i.e., for an original program variable x , the latest version of x will be represented by $x_{(\mathcal{V}[x])}$

We also need to be able to represent the version of a particular assertion A or expression e , after all variables have been replaced with their current versions. We write $A_{\mathcal{V}}$ for this (i.e., $A_{\mathcal{V}} = A[x \mapsto x_{(\mathcal{V}[x])}]$ where $\vec{x} = FV(A)$, and similarly for $e_{\mathcal{V}}$). Note that $x_{\mathcal{V}} = x_{(\mathcal{V}[x])}$ by definition. As in the lectures, we write $FV(s)$ to denote the set of (free; i.e. program variables rather than quantified variables in loop invariants) variables occurring in the statement s . Similarly, we write $mods(s)$ for the set of program variables occurring as the left-hand-side of an assignment statement in s . We use notation \vec{s} to indicate repetition of a particular statement (the variable ranged over in the repetition is left implicit).

Our definition for $toDSA$ is then as follows:

$$\begin{aligned}
toDSA(skip, \mathcal{V}) &= (assume\ true, \mathcal{V}) \\
toDSA(havoc\ x, \mathcal{V}) &= (assume\ true, \mathcal{V}[x \mapsto \mathcal{V}[x] + 1]) \\
toDSA(x := e, \mathcal{V}) &= (assume\ x_{(\mathcal{V}'[x])} = e_{\mathcal{V}}, \mathcal{V}') \text{ where } \mathcal{V}' = \mathcal{V}[x \mapsto \mathcal{V}[x] + 1] \\
toDSA(assume\ A, \mathcal{V}) &= (assume\ A_{\mathcal{V}}, \mathcal{V}) \\
toDSA(assert\ A, \mathcal{V}) &= (assert\ A_{\mathcal{V}}, \mathcal{V}) \\
toDSA(s_1; s_2, \mathcal{V}) &= (s'_1; s'_2, \mathcal{V}_2) \text{ where} \\
&\quad (s'_1, \mathcal{V}_1) = toDSA(s_1, \mathcal{V}) \\
&\quad (s'_2, \mathcal{V}_2) = toDSA(s_2, \mathcal{V}_1) \\
toDSA(s_1 [] s_2, \mathcal{V}) &= ((s'_1; \overrightarrow{y(\mathcal{V}_2[y]) := y(\mathcal{V}_1[y])}) [] (s'_2; \overrightarrow{z(\mathcal{V}_1[z]) := z(\mathcal{V}_2[z])}), \\
&\quad \max(\mathcal{V}_1, \mathcal{V}_2)) \text{ where} \\
&\quad (s'_1, \mathcal{V}_1) = toDSA(s_1, \mathcal{V}) \\
&\quad (s'_2, \mathcal{V}_2) = toDSA(s_2, \mathcal{V}) \\
&\quad \{\overrightarrow{y}\} = \{y \in FV(s_1) \mid \mathcal{V}_2[y] > \mathcal{V}_1[y]\} \\
&\quad \{\overrightarrow{z}\} = \{z \in FV(s_2) \mid \mathcal{V}_1[z] > \mathcal{V}_2[z]\} \\
toDSA(if(b)\{s_1\} else \{s_2\}, \mathcal{V}) &= toDSA((assume\ b; s_1) [] (assume\ \neg b; s_2), \mathcal{V}) \\
toDSA(while(b) invariant A \{s\}, \mathcal{V}) &= (assert\ A_{\mathcal{V}}; (s' [] assume\ (A \wedge \neg b)_{\mathcal{V}}), \mathcal{V}') \\
\text{where} \\
\{\overrightarrow{x}\} &= mods(s) \\
\mathcal{V}' &= \mathcal{V}[x \mapsto \mathcal{V}[x] + 1] \\
(s', -) &= toDSA(assume\ A \wedge b; s; assert\ A; assume\ false, \mathcal{V}')
\end{aligned}$$

In the $s_1 [] s_2$ case, the sequence of extra assignment statements per block is used to “catch up” the versions of any variables which are smaller than the last version used in the other block. For while loops we avoid this step; since the first branch of the introduced non-deterministic choice is guaranteed to end in an *assume false* statement, there is no need to worry about the values of these variables when considering the code after the branches.

Note that at the beginning of each non-deterministic branch, we use the map \mathcal{V}' rather than \mathcal{V} ; this “bakes in” the havoc of written variables from the definition in the lectures: we give each of these variables a new version before hitting the non-deterministic branches which reflect the checking of the loop body and the state after the loop. The reason we don’t simply desugar the loop directly according to the definition from the lectures is that we want to avoid using the standard case for $toDSA$ of a non-deterministic choice, given the optimisation explained in the previous paragraph.