ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 6: Weakest Preconditions

Assignment 1 (Strongest Postconditions)

- $sp(A, x := e) = \exists y.x = e[y/x] \wedge A[y/x]$. Here y is fresh.
- $sp(A, s_1; s_2) = sp(sp(A, s_1), s_2)$.
- $sp(A, \mathit{if}\,(b)\{s_1\}\ \mathit{else}\,\{s_2\}) = sp(b \wedge A, s_1) \vee sp(\neg b \wedge A, s_2)$.
- $sp(A, \mathit{assume}\,A_1) = A \wedge A_1$.
- $sp(A, \mathit{assert}\,A_1) = A \wedge A_1$. Side condition: $A \Rightarrow A_1$.

For a verifier based on strongest postconditions, it is not enough to check only that $sp(A, s)$ implies the program's postcondition while it is enough for a verifier based on weakest libre preconditions to check that the *wlp* is implied by the program's precondition. The reason is that the strongest postcondition expresses what assertion would hold after the program if we reached that point, while the weakest precondition also includes the conditions needed to reach end of the program (or run forever in the case of weakest liberal preconditions). Therefore, a sound verifier based on strongest postconditions would also need to check the side conditions generated by the *assert* statements.

Assignment 2 (Desugaring If-Conditions)

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

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

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 $x_{(0)}, x_{(1)}, x_{(2)}, \ldots$), 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;
if (y > 4) {
    x := x - 1;
} else {
    x := x + 1;
    x := x + 1;
}
assert x > 1;
```

could be converted to

```
x0 := 3;
if (y0 > 4) {
    x1 := x0 - 1;
    x2 := x1;
} else {
    x1 := x0 + 1;
    x2 := x1 + 1;
}
assert x2 > 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:

$$\max(\mathcal{V}_1, \mathcal{V}_2)[i] = \begin{cases} \mathcal{V}_1[i] & \textit{if } \mathcal{V}_1[i] \geq \mathcal{V}_2[i] \\ \mathcal{V}_2[i] & \textit{otherwise} \end{cases}$$

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., $A_{\mathcal{V}} = A[\overrightarrow{x \mapsto x_{(\mathcal{V}[x])}}]$ *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 $\overrightarrow{s}$ 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{array}{rcl}
\textit{toDSA}(\textit{skip}, \mathcal{V}) & = & (\textit{assume true}, \mathcal{V}) \\
\textit{toDSA}(\textit{havoc } x, \mathcal{V}) & = & (\textit{assume true}, \mathcal{V}[x \mapsto \mathcal{V}[x] + 1]) \\
\textit{toDSA}(x{:=}e, \mathcal{V}) & = & (\textit{assume } x_{(\mathcal{V}'[x])} = e_{\mathcal{V}}, \mathcal{V}') \textit{ where } \mathcal{V}' = \mathcal{V}[x \mapsto \mathcal{V}[x] + 1] \\
\textit{toDSA}(\textit{assume } A, \mathcal{V}) & = & (\textit{assume } A_{\mathcal{V}}, \mathcal{V}) \\
\textit{toDSA}(\textit{assert } A, \mathcal{V}) & = & (\textit{assert } A_{\mathcal{V}}, \mathcal{V}) \\
\textit{toDSA}(s_1; s_2, \mathcal{V}) & = & (s_1'; s_2', \mathcal{V}_2) \textit{ where} \\
 & & \quad (s_1', \mathcal{V}_1) = \textit{toDSA}(s_1, \mathcal{V}) \\
 & & \quad (s_2', \mathcal{V}_2) = \textit{toDSA}(s_2, \mathcal{V}_1) \\
\textit{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])}}), \\
 & & \textit{max}(\mathcal{V}_1, \mathcal{V}_2)) \textit{ where} \\
 & & \quad (s_1', \mathcal{V}_1) = \textit{toDSA}(s_1, \mathcal{V}) \\
 & & \quad (s_2', \mathcal{V}_2) = \textit{toDSA}(s_2, \mathcal{V}) \\
 & & \quad \{\overrightarrow{y}\} = \{y \in \textit{FV}(s_1) \mid \mathcal{V}_2[y] > \mathcal{V}_1[y]\} \\
 & & \quad \{\overrightarrow{z}\} = \{z \in \textit{FV}(s_2) \mid \mathcal{V}_1[z] > \mathcal{V}_2[z]\} \\
\textit{toDSA}(\textit{if } (b) \{s_1\} \textit{ else } \{s_2\}, \mathcal{V}) & = & \textit{toDSA}((\textit{assume } b; s_1) \,[]\, (\textit{assume } \neg b; s_2), \mathcal{V}) \\
\textit{toDSA}(\textit{while } (b) \textit{ invariant } A \{s\}, \mathcal{V}) & = & (\textit{assert } A_{\mathcal{V}}; (s' \,[]\, \textit{assume } (A \wedge \neg b)_{\mathcal{V}'}), \mathcal{V}') \\
\multicolumn{3}{l}{\textit{where}} \\
\multicolumn{3}{l}{\quad \{\overrightarrow{x}\} = \textit{mods}(s)} \\
\multicolumn{3}{l}{\quad \mathcal{V}' = \mathcal{V}[\overrightarrow{x \mapsto \mathcal{V}[x] + 1}]} \\
\multicolumn{3}{l}{\quad (s', \_) = \textit{toDSA}(\textit{assume } A \wedge b; s; \textit{assert } A; \textit{assume false}, \mathcal{V}')}
\end{array}
$$

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.