ETH zürich

Alexander J. Summers

Program Verification

Exercise Sheet 6: Hoare Logic and Weakest Preconditions

Assignment 1 (Alternative Hoare Logic Rules)

The lecture slides present alternative Hoare Logic rules for the *havoc*, *assume* and *assert* commands. For *each of these*:

1. Show that an arbitrary instantiation of the alternative rule can be derived using the usual rules. For example, show that the triple $\{A\}$ *havoc* x $\{\exists y.A[y/x]\}$ is derivable (for any assertion A and variable x) using the usual rules.

2. Show that an arbitrary instantiation of the corresponding usual rule can be derived using the alternative rule plus other usual rules. For example, show that $\{\forall y.A[y/x]\}$ *havoc* x $\{A\}$ is derivable (for any assertion A and variable x) using the alternative (*havoc-alt*) rule with the other usual rules.

Assignment 2 (Desugaring If-Conditions)

Consider the following desugaring of an if-condition: we rewrite *if*$(b)\{s_1\}$*else*$\{s_2\}$ into the program (*assume* b; s_1)[](*assume* $\neg b$; s_2). Show that, for any input postcondition A (and for any b, s_1, s_2), applying the *wlp* operator to these two statements yields equivalent results.

Assignment 3 (Dynamic Single Assignment)

The weakest precondition definitions presented in the lecture can produce exponentially large formulas in some cases. This can be alleviated by converting the program into e.g. *dynamic single assignment* (DSA) form: a program is in DSA form if, in each execution (trace) of the program, each variable gets assigned-to *at most once*. This is a little more permissive than, say, static single assignment; the same variable is allowed to be assigned-to in two exclusive branches of the program.

A program can be converted to DSA form as follows: each original program variable x is replaced with a number of *versions* $x_0, x_1, \ldots$ of that variable. During conversion, we need to keep track of the latest version for each original program variable. We introduce a new version of

variable x whenever in the original program the program variable x gets assigned to or *havoc*ed. When dealing with branches (for *if* or non-deterministic choice) we can allow the versions of variables to evolve separately inside both branches. After the two branches, we have to *merge* the versions of the same variable into one (if any new versions of that variable were introduced in the branch). This can be achieved by adding one more version of each variable, and adding an assignment statement at the end of each branch to assign the latest version in the branch to this new variable. For example, the program $x{:=}3;$ *if* $(y > 4)\{x{:=}x{+}1\}$*else*$\{x{:=}x{+}1\};$ *assert* $x > 1$ would become $x_0{:=}3;$ *if* $(y_0 > 4)\{x_1{:=}x_0 + 1;\ x_2{:=}x_1\}$*else*$\{x_1{:=}x_0 + 1; x_2{:=}x_1\};$ *assert* $x_2 > 1$ (in fact, this example illustrates a further possible optimisation when the "last" versions match up in the two branches - what is it?).

The advantage of a program in DSA form is that assignment statements can be handled differently; there is no need for the substitution employed in the *wlp* definition in the lectures[1]. Instead, for a program in DSA form, we can rewrite all variable assignments as *assume* statements: we replace $x{:=}e$ with *assume* $x_n = e$, where x_n is the next version of the variable x. Similarly, we can replace *havoc* x with just *skip* while again taking a new version of x to continue with.

Write a transformation function *toDSA* which takes an annotated program (of the syntax described in the lectures) as input, and returns a new program which is a valid DSA transformation of the original program. In the process, make your function eliminate variable assignments and *havoc* statements (as described here), *if* statements as hinted at in Assignment 2, and *while* loops as in lecture slide 148. Can you also eliminate *skip* from the necessary statement constructs?

[1] Furthermore, once substitutions are no-longer made during weakest-precondition calculations, any duplicated formulas (e.g. in the rule for non-deterministic choice) can be factored out using additional propositional variables, as in the Tseitin CNF transformation on Sheet 1.