Alexander J. Summers

Program Verification

Exercise Sheet 6: Weakest Preconditions

Assignment 1 (Strongest Postconditions)

During the lecture weakest preconditions were presented, which have a dual concept called *strongest postconditions*. Let *sp*(A, s) be a strongest postcondition function where A is the intended precondition and s is the statement in question. We would like the function *sp*(A, s) to satisfy the same requirements that are specified on the slide 146 for *wlp*(s, A):

- *Soundness*: for all s and A, it is guaranteed that $\vDash \{A\} s \{sp(A, s)\}$.
- *Maximality*: for all s, A_1, A_2, if $\vDash \{A_1\} s \{A_2\}$ then $sp(A_1, s) \vDash A_2$.
- *Computability*: for all s and A, *sp*(A, s) is computable.

By using the trace semantics shown on slides 142 and 143 define:

- $sp(A, x := e) = ?$
- $sp(A, s_1; s_2) = ?$
- $sp(A, \mathit{if}\,(b)\{s_1\}\ \mathit{else}\,\{s_2\}) = ?$
- $sp(A, \mathit{assume}\, A_1) = ?$
- $sp(A, \mathit{assert}\, A_1) = ?$

Hint: the strongest postcondition of the assignment statement is very different from its weakest precondition. Do you see why?

Can your definitions of *sp*(A, s) be used to build a sound verifier? If not, what do you need to check in addition to the strongest postcondition to make the verifier sound?

Assignment 2 (Desugaring If-Conditions)

Consider the following desugaring of an if-condition: we rewrite $\mathit{if}(b)\{s_1\}\mathit{else}\{s_2\}$ into the program $(\mathit{assume}\ b;\ s_1)[\,](\mathit{assume}\ \neg b;\ s_2)$. 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)

Recall that the weakest precondition definitions presented in the lecture can produce exponentially large formulas in some cases, which can be alleviated by converting the program into 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:

---

[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.

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, and *if* statements, *while* loops, and *skip* statements as described in the lecture slides.