Semantics of Programming Languages
Exercise Sheet 9

Exercise 9.1  Hoare Logic

In this exercise, you shall prove correct some Hoare triples.
First, write a program that stores the maximum of the values of variables $a$ and $b$ in variable $c$.

definition MAX :: com where

For the next task, you will need the following lemmas. Hint: Sledgehammering may be a good idea.

lemma [simp]: “$(a::int)<b \implies \max a b = b$”
lemma [simp]: “$\neg(a::int)<b \implies \max a b = a$”

Show that $MAX$ satisfies the following Hoare-triple:

lemma “$\vdash$ {λs. True} MAX {λs. s "c" = max (s "a") (s "b")}”

Now define a program $MUL$ that returns the product of $x$ and $y$ in variable $z$. You may assume that $y$ is not negative.

definition MUL :: com where

Prove that $MUL$ does the right thing.

lemma “$\vdash$ {λs. 0 ≤ s "y"} MUL {λs. s "z" = s "x" * s "y"}”

Hints  You may want to use the lemma $\text{algebra.simps}$, that contains some useful lemmas like distributivity.

Note that we use a backward assignment rule. This implies that the best way to do proofs is also backwards, i.e., on a semicolon $S_1; S_2$, you first continue the proof for $S_2$, thus instantiating the intermediate assertion, and then do the proof for $S_1$. However, the first premise of the $\text{Semi}$-rule is about $S_1$. Hence, you may want to use the $\text{rotated}$-attribute, that rotates the premises of a lemma:

lemmas Semi_bwd = Semi[rotated]
Note that our specifications still have a problem, as programs are allowed to overwrite arbitrary variables.

For example, regard the following (wrong) implementation of MAX:

definition "MAX\_wrong ≡ ̄\texttt{a}::=N 0; ̄\texttt{b}::=N 0; ̄\texttt{c}::=N 0"

Prove that MAX\_wrong also satisfies the specification for MAX:

What we really want to specify is, that MAX computes the maximum of the values of a and b in the initial state. Moreover, we may require that a and b are not changed.

For this, we can use logical variables in the specification. Prove the following more accurate specification for MAX:

lemma  \( \vdash \{ \lambda s. a = s \texttt{'}a\}' \land b = s \texttt{'}b\}' \)

\( \vdash \{ \lambda s. \texttt{'}c' = \max a b \land a = s \texttt{'}a' \land b = s \texttt{'}b' \} \)

The specification for MUL has the same problem. Fix it!

**Homework 9  Available Expressions**

*Submission until Wednesday, 11 January 2012, 12:00 (noon).*

Regard the following function AA, which computes the available assignments of a command. An available assignment is a pair of a variable and an expression such that the variable holds the value of the expression in the current state. The function AA c A computes the available assignments after executing command c, assuming that A is the set of available assignments for the initial state.

Note that available assignments can be used for program optimization, by avoiding recomputation of expressions whose value is already available in some variable.

fun AA :: "com ⇒ (vname × aexp) set ⇒ (vname × aexp) set" where

"AA SKIP A = A" |
"AA (x ::= a) A = (if x ∈ vars a then \{} else \{}(x, a)\)} |
\{\(x', a'\). (x', a') ∈ A \land x \notin \{x'\} \cup vars a'}" |
"AA (c\_1; c\_2) A = (AA c\_2 ◦ AA c\_1) A" |
"AA (IF b THEN c\_1 ELSE c\_2) A = AA c\_1 A ∩ AA c\_2 A" |
"AA (WHILE b DO c) A = A ∩ AA c A"

Show that available assignment analysis is a gen/kill analysis, i.e., define two functions gen and kill such that

\( AA c A = (A \cup gen c) − kill c \).

Note that the above characterization differs from the one that you have seen on the slides, which is \((A − kill c) \cup gen c\). However, the same properties (monotonicity, etc.) can be derived using either version.
fun gen :: “com ⇒ (vname × aexp) set”
and “kill” :: “com ⇒ (vname × aexp) set”

lemma AA_gen_kill: “AA c A = (A ∪ gen c) − kill c”

Hint: Defining gen and kill functions for available assignments will require mutual recursion, i.e., gen must make recursive calls to kill, and kill must also make recursive calls to gen. The and-syntax in the function declaration allows you to define both functions simultaneously with mutual recursion. After the where keyword, list all the equations for both functions, separated by | as usual.

Now show that the analysis is sound:

theorem AA_sound:
“(c, s) ⇒ s’ ⟹ ∀(x, a) ∈ AA c {}. s’ x = aval a s’”

Hint: You will have to generalize the theorem for the induction to go through.