Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Brian Huffman, Peter Lammich

WS 2011/12
16. 12. 2011

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*]: "$(a::int)<b \Longrightarrow max\ a\ b = b$"

lemma [*simp*]: "$\neg(a::int)<b \Longrightarrow max\ a\ b = a$"

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

lemma "$\vdash \{\lambda s.\ True\}\ MAX\ \{\lambda s.\ s\ ''c'' = max\ (s\ ''a'')\ (s\ ''b'')\}$"

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 "$\vdash \{\lambda s.\ 0 \leq s\ ''y''\}\ MUL\ \{\lambda s.\ s\ ''z'' = s\ ''x'' * s\ ''y''\}$"

Hints You may want to use the lemma *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 *Semi*-rule is about S_1. Hence, you may want to use the *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 ≡ ''a''::=N 0; ''b''::=N 0; ''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 "⊢ {λs. a=s ''a'' ∧ b=s ''b''}
  MAX
  {λs. s ''c'' = max a b ∧ a = s ''a'' ∧ b = s ''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 ∧ x ∉ {x'} ∪ vars a'}" |
  "AA (c₁; c₂) A = (AA c₂ ∘ AA c₁) A" |
  "AA (IF b THEN c₁ ELSE c₂) A = AA c₁ A ∩ AA c₂ 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* $\Rightarrow$ *(vname* $\times$ *aexp) set"*
and *"kill"* :: *"com* $\Rightarrow$ *(vname* $\times$ *aexp) set"*

lemma *AA_gen_kill*: *"AA c A* $=$ *(A* $\cup$ *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*:
  "$(c,\ s) \Rightarrow s' \Longrightarrow \forall (x,\ a) \in AA\ c\ \{\}.\ s'\ x = aval\ a\ s'$"

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