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

WS 2015/16
15. 12. 2015

Semantics of Programming Languages

Exercise Sheet 10

Exercise 10.1 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*]: "$\neg(a::int)<b \Longrightarrow max\ a\ b = a$"
by *auto*

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 *Seq*-rule is about S_1. Hence, you may want to use the *rotated*-attribute, that rotates the premises of a lemma:

lemmas *Seq_bwd* = *Seq*[*rotated*]

lemmas *hoare_rule[intro?]* = *Seq_bwd Hoare.Assign Assign' If*

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!

Exercise 10.2 Denotational Semantics

Define a denotational semantics for REPEAT-loops, and show its equivalence to the bigstep semantics.

Use the exercise template that we provide on the course web page.

Homework 10.1 Floyd's Method for Program Verification

Submission until Tuesday, Dec 22, 10:00am.

A flow graph is a directed graph with labeled edges. Labels come with an enabled predicate and an effect function. The enabled predicate checks whether a label is enabled in a state, and the effect function applies the effect of a label to a state.

The following formalizes this setting:

type_synonym *('n,'l) flowgraph* = *"'n ⇒ 'l ⇒ 'n ⇒ bool"*

locale *flowgraph* =
fixes *G* :: *"('n,'l) flowgraph"*
fixes *enabled* :: *"'l ⇒ 's ⇒ bool"*
fixes *effect* :: *"'l ⇒ 's ⇒ 's"*
begin

Define a small-step semantics on flow graphs: Configurations are pairs of nodes and states. A step is induced by an enabled edge, and applies the effect of the edge to the state.

inductive *step* :: "('n×'s) ⇒ ('n×'s) ⇒ bool"

We form the reflexive transitive closure over our small-step semantics:

abbreviation *"steps ≡ star step"*

The idea of Floyd's method is to annotate an invariant over states to each node in the flow graph, and show that the invariant is preserved by the edges:

context
  fixes *I* :: "'n ⇒ ('s ⇒ bool)"
  assumes *preserve*: "⟦ I n s; G n l n'; enabled l s ⟧ ⟹ I n' (effect l s)"
begin

Show that the invariant is preserved by multiple steps:

  lemma *preserves*:
    assumes *"I n s"*
    assumes *"steps (n,s) (n',s')"*
    shows *"I n' s'"*

end
end

Now, let's instantiate the flow graph framework for IMP-programs. Edges are labeled by conditions, assignments, or skip.

datatype *label = Assign vname aexp | Cond bexp | Skip*

Define the enabled and effect functions for edges:

fun *enabled* :: *"label ⇒ state ⇒ bool"*

fun *effect* :: *"label ⇒ state ⇒ state"*

For nodes, we use commands. Similar to the small-step semantics, a node indicates the command that still has to be executed. Define the flow graph for IMP programs. We give you the case for assignment and if-false here, you have to define the other cases. Make sure that you use the same intermediate steps as *op* → does, this will simplify the next proof:

inductive *cfg* :: *"com ⇒ label ⇒ com ⇒ bool"*
where
  "cfg (x::=a) (Assign x a) SKIP"
  | *"cfg (IF b THEN c1 ELSE c2) (Cond (Not b)) c2"*

The following instantiates the flow graph framework:

interpretation *flowgraph cfg enabled effect* **.**
term *step* **term** *steps*

Show that execution of flow graphs and the small-step semantics coincide:

lemma *steps_eq*: "$cs \rightarrow* cs' \longleftrightarrow steps\ cs\ cs'$"

Combine your results to prove the following theorem, which allows you to prove correctness of programs with Floyd's method. (Hint: Big and small-step semantics are equivalent!)

lemma *floyd*:
  assumes *PRE*: "$\bigwedge s.\ P\ s \Longrightarrow I\ c\ s$"
  assumes *PRES*: "$\bigwedge n\ s\ c\ l\ c'.\ [\![cfg\ c\ l\ c';\ I\ c\ s;\ enabled\ l\ s\ ]\!] \Longrightarrow I\ c'\ (effect\ l\ s)$"
  assumes *POST*: "$\bigwedge s.\ I\ SKIP\ s \Longrightarrow Q\ s$"
  shows "$\models \{P\}\ c\ \{Q\}$"

Homework 10.2 Application of Floyd's Method

Submission until Tuesday, Dec 22, 10:00am. **5 bonus points, quite hard**

Apply Floyd's method to verify the following program:

definition "*P* ≡
  ''r''::= N 0;;
  WHILE (Less (N 0) (V ''x'')) DO (
    ''r'' ::= Plus (N 2) (V ''r'');;
    ''x'' ::= Plus (N (−1)) (V ''x'')
)"

lemma "$\models \{\ \lambda s.\ s\ ''x'' = x \wedge x \geq 0\ \}\ P\ \{\lambda s.\ s\ ''r'' = 2{*}x\}$"

Hints:

- You have to define an appropriate invariant for each reachable node in the control flow graph. Define the invariants for unreachable nodes to be false on all states.
- Use abbreviations for parts of the program to simplify writing the reachable nodes.
- Try use automation, in particular to identify unreachable nodes and discharge the vacuous proof obligations resulting from assuming invariants of unreachable nodes.
- If necessary, use smaller programs to get a feeling for using this proof technique.