Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2021/22
21.12.2021

Semantics of Programming Lectures

Exercise Sheet 09

Exercise 9.1 Hoare Logic

In this exercise, you shall prove correct some Hoare triples.

Step 1 Write a program that stores the maximum of the values of variables a and b in variable c.

definition *Max* :: *"com"*

Step 2 Prove these lemmas about *max*:

lemma *max_right*[*simp*]: *"(a::int)<b ⟹ max a b = b"*

lemma *max_left*[*simp*]: *"¬(a::int)<b ⟹ max a b = a"*

Show that *Tut.Max* satisfies the following Hoare triple:

lemma *"⊢ {λs. True} Max {λs. s ''c'' = max (s ''a'') (s ''b'')}"*

Step 3 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"*

Step 4 Prove that *MUL* does the right thing.

lemma *"⊢ {λs. 0 ≤ s ''y''} MUL {λs. s ''z'' = s ''x'' * s ''y''}"*

Hints:

- You may want to use the lemma *algebra_simps*, containing 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 $c_1;;\ c_2$, you first continue the proof for c_2, thus instantiating the intermediate assertion, and then do the proof for c_1. However, the first premise of the *Seq*-rule is about c_1. In an Isar proof, this is no problem. In an **apply**-style proof, the ordering matters. Hence, you may want to use the [*rotated*] attribute:

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

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

Step 5 Note that our specifications still have a problem, as programs are allowed to overwrite arbitrary variables.
For example, regard the following (wrong) implementation of *Tut.Max*:

definition "*MAX_wrong* = (''a''::=*N* 0;;''b''::=*N* 0;;''c''::= *N* 0)"

Prove that *MAX_wrong* also satisfies the specification for *Tut.Max*:

lemma "⊢ {λs. *True*} *MAX_wrong* {λs. s ''c'' = *max* (s ''a'') (s ''b'')}"

What we really want to specify is, that *Tut.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 *Tut.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 9.2 Forward Assignment Rule

Think up and prove correct a forward assignment rule, i.e., a rule of the form $\vdash \{P\}\ x ::= a\ \{Q\}$, where Q is some suitable postcondition. Hint: To prove this rule, use the completeness property, and prove the rule semantically.

lemmas *fwd_Assign'* = *weaken_post*[*OF fwd_Assign*]

Redo the proofs for *Tut.Max* and *MUL* from the previous exercise, this time using your forward assignment rule.

lemma "⊢ {λs. *True*} *Max* {λs. s ''c'' = *max* (s ''a'') (s ''b'')}"

lemma "⊢ {λs. 0 ≤ s ''y''} *MUL* {λs. s ''z'' = s ''x'' * s ''y''}"

Homework 9.1 Floyd's Method for Program Verification

Submission until Sunday, Jan 09, 23:59pm.

This homework runs in parallel to the be original project and is worth 10 points (but less effort than a regular sheet).

A flow graph is a directed graph with labeled edges:

type_synonym $('n,'l)$ *flowgraph* = "$'n \Rightarrow 'l \Rightarrow 'n \Rightarrow bool$"

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 by defining a context in which G, *enabled*, and *effect* are fixed (but arbitrary) constants:

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

We 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. Define the single *step* rule:

inductive *step* :: "$('n \times 's) \Rightarrow ('n \times 's) \Rightarrow bool$"

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

abbreviation "*steps* $\equiv$ *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. Again, we use a context to fix an invariant and assume preservation:

context
  fixes I :: "$'n \Rightarrow ('s \Rightarrow bool)$"
  assumes *preserve*: "$\llbracket\ I\ n\ s;\ G\ n\ l\ n';\ enabled\ l\ s\ \rrbracket \Longrightarrow I\ n'\ (effect\ l\ s)$"
begin

Show that the invariant is preserved by multiple steps:

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

end

end

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

datatype *label* = *Assign* (*char list*) *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 ($\rightarrow$) 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"*

code_pred *cfg* **.**

The following instantiates the flow graph context:

interpretation *flowgraph cfg enabled effect* **.**

I.e., you can now use the *preserves* lemma outside its context:

thm *preserves*

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

lemma *steps_eq*: *"cs →∗ cs′ ⟷ 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*: *"⋀s. P s ⟹ I c s"*
    and *PRES*: *"⋀n s c l c′. ⟦cfg c l c′; I c s; enabled l s ⟧ ⟹ I c′ (effect l s)"*
    and *POST*: *"⋀s. I SKIP s ⟹ Q s"*
  shows *"⊨ {P} c {Q}"*

Homework 9.2 Be Original!

Submission until Sunday, Jan 9, 23:59pm. Think up a nice topic to formalize yourself, for example

- Prove some interesting result about automata/formal language theory
- Formalize some results from mathematics

- Show properties of some interesting algorithm
- ...

In total, this exercise will be worth 20 points, plus bonus points for nice submissions.

Formalize your topic!

- you should set yourself a time limit before you start
- incomplete/unfinished formalizations are welcome and will be graded
- comment your formalization well, such that we can see what it does/is intended to do
- the code quality of your formalization also matters, so make sure to use the linter add-on component

Merry Christmas!