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 ⇒ max a b = b"

lemma [simp]: "¬(a::int)<b ⇒ max a b = a"

by auto

Show that $\text{MAX}$ satisfies the following Hoare-triple:

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

Now define a program $\text{MUL}$ 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 $\text{MUL}$ does the right thing.

lemma "{\lambda s. 0 ≤ s''y''} \text{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 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:

\[
\text{definition } \quad \text{"MAX\_wrong \equiv \text{"a"::=N 0; "b"::=N 0; "c"::=N 0"} }
\]

Prove that \text{MAX\_wrong} also satisfies the specification for \text{MAX}:

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

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

\[
\text{lemma } \quad \vdash \{\lambda s. a=s \text{"a"} \land b=s \text{"b"}\} \\
\text{MAX} \\
\{\lambda s \text{"c"} = \text{max} a b \land a = s \text{"a"} \land b = s \text{"b"}\}
\]

The specification for \text{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:

\[
\text{type_synonym } (\text{"n",\text{"l"}) flowgraph = "\text{"n" \Rightarrow \text{"l" \Rightarrow \text{\text{"n" \Rightarrow bool}))}
\]

\[
\text{locale flowgraph =} \\
\text{fixes } G :: (\text{"n",\text{"l"}) flowgraph"
\text{fixes } enabled :: "\text{"l" \Rightarrow \text{\text{"s" \Rightarrow bool))}
\text{fixes } effect :: "\text{"l" \Rightarrow \text{\text{"s" \Rightarrow \text{"s"))}
\text{begin}
\]

2
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.

\textbf{inductive} \textit{step} :: \(\langle n \times s \rangle \Rightarrow \langle n' \times s' \rangle \Rightarrow \text{bool}\)

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

\textbf{abbreviation} \textit{steps} \equiv \textit{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:

\begin{verbatim}
context
  fixes \(I \::\langle n \Rightarrow \langle s \Rightarrow \text{bool} \rangle \rangle\)
  assumes \textit{preserve}: \[ I n s; G n l n'; enabled l s ] \Rightarrow I n' (effect l s)

begin

Show that the invariant is preserved by multiple steps:

\textbf{lemma} \textit{preserves}:
  assumes \textit{I n s}
  assumes \textit{steps (n,s) (n',s')} 
  shows \textit{I n' s'}

end

end
\end{verbatim}

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

\textbf{datatype} \textit{label} = \textit{Assign} \textit{vname aexp} \mid \textit{Cond} \textit{bexp} \mid \textit{Skip}

Define the enabled and effect functions for edges:

\textbf{fun} \textit{enabled} :: \textit{label} \Rightarrow \textit{state} \Rightarrow \textit{bool}

\textbf{fun} \textit{effect} :: \textit{label} \Rightarrow \textit{state} \Rightarrow \textit{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 \(op \rightarrow \text{do}\), this will simplify the next proof:

\textbf{inductive} \textit{cfg} :: \textit{com} \Rightarrow \textit{label} \Rightarrow \textit{com} \Rightarrow \textit{bool}

\textbf{where}
  \textit{cfg} \((x::a) (Assign x a) \text{SKIP}\)
  \mid \textit{cfg} \((\text{IF} \: b \: \text{THEN} \: c1 \: \text{ELSE} \: c2) (\text{Cond} (\text{Not} \: b)) \: c2\)

The following instantiates the flow graph framework:
Show that execution of flow graphs and the small-step semantics coincide:

**Lemma steps_eq:** \( \text{cs} \rightarrow \ast \text{cs}' \iff \text{steps} \text{cs cs}' \)

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:** \( \forall s. \ P \ s \implies I \ c \ s \)
- **Assumes PRES:** \( \forall n \ s \ c \ l \ c'. [\text{cfg} \ c \ l \ c'; I \ c \ s; \text{enabled} \ l \ s] \implies I \ c' (\text{effect} \ l \ s) \)
- **Assumes POST:** \( \forall s. I \text{SKIP} \ s \implies Q \ s \)
- **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 ≡\\n  "r"::= N 0;;
  "WHILE (Less (N 0) (V "x")) DO (\\n  "r"::= Plus (N 2) (V "r"));;
  "x"::= Plus (N (−1)) (V "x")
)"

**Lemma** \( \models \{ \lambda s. \ "x" = x \land x \geq 0 \} \ P \ \{\lambda s. \ "r" = 2s\times\} \)

**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.