Semantics of Programming Languages
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 $\text{max}$:

lemma [simp]: "((a::int)<b) \Rightarrow \text{max} a b = b"
lemma [simp]: "\neg((a::int)<b) \Rightarrow \text{max} a b = a"

Show that $ex09.Max$ satisfies the following Hoare triple:

lemma "\vdash \{\lambda s. \text{True} \} \text{Max} \{\lambda s. s''c'' = \text{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 where

Step 4  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, 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 $\text{Seq}$-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 $\text{Seq.bwd} = \text{Seq[rotated]}$

lemmas hoare_rule[intro?] = $\text{Seq.bwd Assign Assign'}$ 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 $\text{ex09.Max}$:

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

Prove that $\text{MAX_wrong}$ also satisfies the specification for $\text{ex09.Max}$:
lemma "\{\lambda s. True\} MAX_wrong \{\lambda s. s''c'' = \max (s ''a'') (s ''b'')\}"

What we really want to specify is, that $\text{ex09.Max}$ 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 $\text{ex09.Max}$:
lemma "\{\lambda s. a=s ''a'' \land b=s ''b''\} 
MAX \{\lambda s. s''c'' = \max a b \land a = s ''a'' \land b = s ''b''\}"

The specification for $\text{MUL}$ 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 $\text{ex09.Max}$ and $\text{MUL}$ from the previous exercise, this time using your forward assignment rule.

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

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

**Homework 9.1 Fixed point reasoning**

Submit until Sunday, Jan 17, 23:59.

In the course, you have seen the Knaster-Tarski least fixed point theorem. The relevant constant is $\text{lfp} :: \ ('a \Rightarrow 'a) \Rightarrow 'a$, which assumes a complete lattice order $\leq$ on $'a$ and returns, for each monotonic operator $f :: 'a \Rightarrow 'a$, its least fixed point $\text{lfp} f$. 

2
So far, we've only dealt with the case where \( a \) is \( b \) set (the type of sets over an arbitrary type \( b \)) and \( \leq \) is \( \subseteq \) (set inclusion). In this exercise, you will prove a different kind of fixed point theorem. It says that if there are two injective functions, one from \( a \) to \( b \), and one the other way round, then there also exists an bijection between \( a \) and \( b \):

**Theorem**

**Assumes** "inj \((\text{f} :: a \Rightarrow b)\)" and "inj \((\text{g} :: b \Rightarrow a)\)"

**Shows** "\( \exists h :: a \Rightarrow b. \ inj h \land surj h \)"

This is a fixed point theorem because we will use a least fixed point for the construction of \( h \). Follow the proof outline below to finish the proof.

**Theorem fixp**

**Assumes** "inj \((\text{f} :: a \Rightarrow b)\)" and "inj \((\text{g} :: b \Rightarrow a)\)"

**Shows** "\( \exists h :: a \Rightarrow b. \ inj h \land surj h \)"

**Proof**

Define \( S \) where \( S \equiv \text{lfp} (\lambda X. - (g' (- (f' X)))) \)

Let \( ?g' = \text{inv g} \)

Define \( h \) where \( h \equiv \lambda z. \text{if } z \in S \text{ then f z else } ?g' z \)

Have "\( S = - (g' (- (f' S))) \)"

Have *: "\( ?g' (- S) = - (f' S) \)"

Show "\( \text{inj } h \land \text{surj } h \)"

Proof

From * show "surj h"

Have "inj on f S"

Moreover have "inj on ?g' (- S)"

Moreover { fix a b assume "a \in S" "b \in - S" and eq: "f a = ?g' b"

Have False

} ultimately show "inj h"

qed

Homework 9.2  A Hoare Calculus with Execution Times

Submission until Sunday, Jan 17, 23:59.

In this homework, we will consider a Hoare calculus with execution times.

**Step 1** We first give a modified big-step semantics to account for execution times. A judgement of the form \((c, s) \Rightarrow \text{"n } t\) has the intended meaning that we can get from state \( s \) to state \( t \) by an terminating execution of program \( c \) that takes exactly \( n \) time steps.
inductive
big_step :: "com × state ⇒ nat ⇒ state ⇒ bool" ("_⇒_" 55)
where
Skip: "(SKIP, s) ⇒ s1 s" |
Assign: "(x ::= a, s) ⇒ s1 s(x := aval a s)" |
Seq: "[(c1,s1) ⇒ s1 s2; (c2,s2) ⇒ s2 s3; n1+n2 = n3] ⇒ (c1;c2, s1) ⇒ s3 s4" |
IfTrue: "[bv b s; (c1,s) ⇒ s1 t; n3 = Suc n1] ⇒ (IF b THEN c1 ELSE c2, s) ⇒ s3 t" |
IfFalse: "[¬bv b s; (c2,s) ⇒ s2 t; n3 = Suc n2] ⇒ (IF b THEN c1 ELSE c2, s) ⇒ s3 t" |
WhileFalse: "[¬bv b s] ⇒ (WHILE b DO c, s) ⇒ s1 s" |
WhileTrue: "[bv b s1; (c,s) ⇒ s1 s2; (WHILE b DO c, s2) ⇒ s2 s3; n1+n2+1 = n3] ⇒ (WHILE b DO c, s1) ⇒ s3 s4"

Step 2 Some theoretical background: We need extended natural numbers. These are provided by the HOL−Library.Extended_Nat theory. We can imagine extended natural numbers as the union of all natural numbers \( \mathbb{N} \) and \( \infty \). Here are some examples to illustrate their arithmetic behaviour:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3::enat</td>
<td>3</td>
</tr>
<tr>
<td>∞::enat</td>
<td>( \infty )</td>
</tr>
<tr>
<td>(3::enat) + 4</td>
<td>7</td>
</tr>
<tr>
<td>(3::enat) + ∞</td>
<td>( \infty )</td>
</tr>
<tr>
<td>eSuc 3</td>
<td>4</td>
</tr>
<tr>
<td>eSuc ∞</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

Step 3 Next, we define a Hoare calculus that also accounts for execution times. Assertions are still the same (of type state ⇒ bool), but we introduce new quantitative assertions of type state ⇒ enat.

type_synonym assn = "state ⇒ bool"
type_synonym qassn = "state ⇒ enat"

It is thought that the result of a qassn represents a potential, where \( \infty \) corresponds to a False assertion in classical Hoare calculus. We can hence embed assertions into quantitative assertions:

fun emb :: "bool ⇒ enat" ("↓") where
"emb False = ∞"
"emb True = 0"

We can define what it means for a quantitative Hoare triple to be valid:

definition hoare_Qvalid :: "qassn ⇒ com ⇒ qassn ⇒ bool" ("_⇒Q_" 50) where
\(\{P\} \ c \ \{Q\} \iff (\forall s. \ P \ s < ∞ \rightarrow (\exists t. p \ t \land P \ s \geq p + Q \ t))\)
Finally, we define quantitative Hoare judgements. The idea is that both pre- and post-condition assign an \textit{enat} to a state that is then decreased as the execution progresses.

We will see an example in the next step.

\textbf{inductive} \texttt{hoareQ} :: “\texttt{qassn ⇒ com ⇒ qassn ⇒ bool}” (“\(\vdash \texttt{Q} (((\lambda x).)/ (\_)/ (((\lambda x).)))\)” 50) \textbf{where}

- Skipping and assignment both decrease the potential.
  \texttt{SkipQ}: “\(\vdash \texttt{Q} \{\lambda s. \texttt{eSuc} (P s)\} \texttt{SKIP} \{P\} \) |
  \texttt{AssignQ}: “\(\vdash \texttt{Q} \{\lambda s. \texttt{eSuc} (P (s[a/x]))\} x::=a \{P\}\)” |
- \texttt{IF \_ THEN \_ ELSE \_} is a bit tricky: We decrease the potential by one before executing either branch. Then we add 0 to the branch that gets executed and \(\infty\) to the branch that does not get executed. This is similar to how in classical Hoare calculus, the branch that does not get executed gets \texttt{False} as precondition.
  \texttt{IfQ}: “\(\left[\begin{array}{l}
  \vdash \texttt{Q} \{\lambda s. P s + \downarrow(\neg bval b s)\} c_1 \{Q\};
  \vdash \texttt{Q} \{\lambda s. P s + \downarrow(\neg bval b s)\} c_2 \{Q\} \;
  \end{array}\right]\) |
  \(\Rightarrow \vdash \texttt{Q} \{\lambda s. \texttt{eSuc} (P s)\} \texttt{IF} b \texttt{THEN} c_1 \texttt{ELSE} c_2 \{Q\}\)” |
- \texttt{Sequence} works about as expected.
  \texttt{SeqQ}: “\(\vdash \texttt{Q} \{P_1\} c_1 \{P_2\}; \vdash \texttt{Q} \{P_2\} c_2 \{P_3\}\) |
  \(\Rightarrow \vdash \texttt{Q} \{P_1\} c_1;c_2 \{P_3\}\)” |
- \texttt{WHILE \_ DO \_} is a combination of conditional and sequence. The invariant is also a function to \texttt{enat}.
  \texttt{WhileQ}: “\(\vdash \texttt{Q} \{\lambda s. I s + \downarrow(\neg bval b s)\} c \{\lambda t. I t + 1\} \)
  \(\Rightarrow \vdash \texttt{Q} \{\lambda s. I s + 1\} \texttt{WHILE} b \texttt{DO} c \{\lambda s. I s + \downarrow(\neg bval b s)\}\)” |
- The consequence rule also works like in the classic Hoare calculus.
  \texttt{ConseqQ}: “\(\left[\begin{array}{l}
  \vdash \texttt{Q} \{P\} c \{Q\}; \forall s. P s \leq P’ s; \forall s. Q’ s \leq Q s \;
  \end{array}\right]\)
  \(\Rightarrow \vdash \texttt{Q} \{P’\} c \{Q’\}\)”

\textbf{Step 4} To exercise our newly-introduce Hoare calculus with timing, we will prove a Hoare triple for an example program that computes the sum of numbers from 1 to \(n\). However, we are only interested in computing the total runtime and disregard correctness properties.

\textbf{definition} \texttt{wsum} :: \texttt{com} \textbf{where}

\texttt{wsum} =
  "\texttt{y}" ::= \texttt{N 0};;
  \texttt{WHILE} \texttt{Less} \(N 0\) \texttt{(V "\textit{x}")}
  \texttt{DO} ("\texttt{y}" ::= \texttt{Plus} \(V "\texttt{y}"\) \(V "\texttt{x}"\);;
  "\texttt{x}" ::= \texttt{Plus} \(V "\texttt{x}"\) \(N (-1)\))"

The following lemma states the the \texttt{wsum} program will take at most \(2 + 3 \times n\) steps to complete. Prove it!

\textbf{theorem} \texttt{wsum}: “\(\vdash \texttt{Q} \{\lambda s. \texttt{enat} (2 + 3 \times n) + \downarrow (s "\texttt{y}" = \texttt{int} n)\} \texttt{wsum} \{\lambda s. 0\}\)”

\textbf{unfolding} \texttt{wsum_def}

\textbf{apply} (\texttt{rule SeqQ[rotated]})
apply (rule conseqQ)
apply (rule WhileQ [where \( I = \lambda s. \text{enat} (3 \ast \text{nat} (s'')) \)])

**Step 5** You task is to prove a fragment of soundness (without the while case). The SKIP-case is already demonstrated below. Prove the remaining extracted lemmas. You don’t need to prove the final theorem.

**lemma Skip_sound:** \( \models Q \{ \lambda a. \text{eSuc} (P a) \} \text{SKIP} \{ P \} \)

**unfolding hoare_Qvalid_def proof (safe)**

- fix \( s \)
- assume \( \text{eSuc} (P s) < \infty \)
- then have \( \text{(SKIP}, s) \Rightarrow t \in \text{enat} n \ast P t \leq \text{eSuc} (P s) \) using Skip_eSuc_def by (auto split: enat.splits)

- thus \( \exists t n. \text{(SKIP}, s) \Rightarrow \text{enat} n \ast P t \leq \text{eSuc} (P s) \) by blast

qed

**theorem Assign_sound:** \( \models Q \{ \lambda b. \text{eSuc} (P (b[a/x])) \} x::=a \{ P \} \)

**theorem conseq_sound:**

- assumes \( \text{hyps: } \forall s. P s \leq P' s \)
- assumes \( \text{IH: } \models Q \{ P \} c \{ Q \} \)
- shows \( \models Q \{ P' \} c \{ Q' \} \)

**theorem If_sound:**

- assumes \( \models Q \{ \lambda a. P a + \downarrow \text{(beal b a)} \} c_1 \{ Q \} \)
- assumes \( \models \{ \lambda a. P a + \downarrow \neg \text{beal b a} \} c_2 \{ Q \} \)
- shows \( \models Q \{ \lambda a. \text{eSuc} (P a) \} \text{IF} \ b \text{ THEN} \ c_1 \text{ ELSE} \ c_2 \{ Q \} \)

**theorem Seq_sound:**

- assumes \( \models Q \{ P_1 \} c_1 \{ P_2 \} \)
- assumes \( \models Q \{ P_2 \} c_2 \{ P_3 \} \)
- shows \( \models Q \{ P_1 \} c_1;c_2 \{ P_3 \} \)

**theorem hoareQ_sound:** \( \models Q \{ P \} c \{ Q \} \Rightarrow \models Q \{ P \} c \{ Q \} \)

---

**Homework 9.3 Traces (Bonus Exercise)**

*Submission until Sunday, Jan 17, 23:59. This is a bonus exercise worth 4 points.*

In this exercise, we explore a new computational model: event traces.

An event is either an action which has an effect (in our IMP language, an assignment), or a test:

```plaintext
datatype event = Action string aexp | Test bexp
```

A trace is a sequence of events, which corresponds to a computation.

Given an event trace and a starting state, the `exec` function 'replays' the computation.

All of the tests in the event trace should succeed; if one fails, the execution stops:
fun exec :: "state ⇒ event list ⇒ state option" where
exec s [] = Some s |
exec s (Action x a # ts) = exec (s(x := aval a s)) ts |
exec s (Test b # ts) = (if beval b s then exec s ts else None)"

abbreviation "example ≡ [Action "x" (N 1), Test (Less (N 0) (V "x"))]"
value "case (exec <> example) of Some t ⇒ t "x"

We now want to compute the set of possible event traces for a given command. For instance, IF (Bc True) THEN "x"::=(N 1) ELSE SKIP has the traces \{[Test (Bc True), Action "x" (N 1)], [Test (Not (Bc True))]\}. Start by defining an predicate trace, which characterizes traces for a command:

inductive trace :: "com ⇒ event list ⇒ bool"

From this it should be easy to define the set of all possible traces:

abbreviation traces :: "com ⇒ event list set"

Prove that that every big step has a corresponding trace:

theorem big_traces: "(c,s) ⇒ t ⇒ ∃ts ∈ traces c. exec s ts = Some t"

Next, prove the other direction:

theorem trace_big: "[trace c ts; exec s ts = Some t] ⇒ (c,s) ⇒ t"

Finally, the equivalence to big-step semantics follows.

lemma "(c,s) ⇒ t ⇔ (∃ts ∈ traces c. exec s ts = Some t)"
using big_traces trace_big by auto