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

WS 2020/21
11.1.2021

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 *max*:

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

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

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

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* **where**

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

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

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 *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 *ex09.Max*:

definition "$MAX\_wrong = (''a''::=N\ 0;;''b''::=N\ 0;;''c''::=\ N\ 0)$"

Prove that *MAX_wrong* also satisfies the specification for *ex09.Max*:

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

What we really want to specify is, that *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 *ex09.Max*:

lemma "$\vdash \{\lambda s.\ a{=}s\ ''a'' \wedge b{=}s\ ''b''\}$
$Max\ \{\lambda s.\ s\ ''c'' = max\ a\ b \wedge a = s\ ''a'' \wedge b = s\ ''b''\}$"

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 *ex09.Max* and *MUL* from the previous exercise, this time using your forward assignment rule.

lemma "$\vdash \{\lambda s.\ True\}\ Max\ \{\lambda s.\ s\ ''c'' = max\ (s\ ''a'')\ (s\ ''b'')\}$"
lemma "$\vdash \{\lambda s.\ 0 \le s\ ''y''\}\ MUL\ \{\lambda s.\ s\ ''z'' = s\ ''x'' * s\ ''y''\}$"

Homework 9.1 Fixed point reasoning

Submission until Sunday, Jan 17, 23:59.

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

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 (f :: 'a ⇒ 'b)" and "inj (g :: 'b ⇒ 'a)"
  shows "∃ h :: 'a ⇒ 'b. inj h ∧ 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 (f :: 'a ⇒ 'b)" and "inj (g :: 'b ⇒ 'a)"
  shows "∃ h :: 'a ⇒ 'b. inj h ∧ surj h"
proof
  define S where "S ≡ lfp (λX. − (g ` (− (f ` X))))"
  let ?g' = "inv g"
  define h where "h ≡ λz. if z ∈ S then f z else ?g' z"

  have "S = − (g ` (− (f ` S)))"
    have *: "?g' ` (− S) = − (f ` S)"

  show "inj h ∧ surj h"
  proof
    from * show "surj h"
      have "inj_on f S"
      moreover have "inj_on ?g' (− S)"
      moreover {
    fix a b
    assume "a ∈ S" "b ∈ − S" and eq: "f a = ?g' b"
        have False
      }
    ultimately show "inj h"
    qed
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 $(c,\ s) \Rightarrow\hat{}n\ t$ 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_t :: *"com × state ⇒ nat ⇒ state ⇒ bool"* (*"_ ⇒ˆ/_ _" 55*)
where
Skip: *"(SKIP, s) ⇒ˆ1 s"* |
Assign: *"(x ::= a,s) ⇒ˆ1 s(x := aval a s)"* |
Seq: "⟦ $(c_1,s_1) \Rightarrow\hat{\ } n_1\ s_2$; $(c_2,s_2) \Rightarrow\hat{\ } n_2\ s_3$; $n_1+n_2 = n_3$ ⟧ ⟹ $(c_1;;c_2,\ s_1) \Rightarrow\hat{\ } n_3\ s_3$" |
IfTrue: "⟦ *bval b s*; $(c_1,s) \Rightarrow\hat{\ } n_1\ t$; $n_3 = Suc\ n_1$ ⟧ ⟹ (*IF b THEN* c_1 *ELSE* c_2, *s*) $\Rightarrow\hat{\ } n_3\ t$" |
IfFalse: "⟦ ¬*bval b s*; $(c_2,s) \Rightarrow\hat{\ } n_2\ t$; $n_3 = Suc\ n_2$ ⟧ ⟹ (*IF b THEN* c_1 *ELSE* c_2, *s*) $\Rightarrow\hat{\ } n_3\ t$"
|
WhileFalse: "⟦ ¬*bval b s* ⟧ ⟹ (*WHILE b DO c*, *s*) ⇒ˆ*1 s*" |
WhileTrue:
"⟦ *bval b* s_1; $(c,s_1) \Rightarrow\hat{\ } n_1\ s_2$; (*WHILE b DO c*, s_2) $\Rightarrow\hat{\ } n_2\ s_3$; $n_1+n_2+1 = n_3$ ⟧
⟹ (*WHILE b DO c*, s_1) $\Rightarrow\hat{\ } n_3\ s_3$"

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 ∞. Here are some examples to illustrate their arithmetic behaviour:

value *"3::enat"* — 3
value *"∞::enat"* — ∞
value *"(3::enat) + 4"* — 7
value *"(3::enat) + ∞"* — ∞
value *"eSuc 3"* — 4
value *"eSuc ∞"* — ∞

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 ∞ 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"*
("$\models_Q$ {(1_)}/ (_)/ {(1_)}" *50*) **where**
"$\models_Q \{P\}\ c\ \{Q\} \longleftrightarrow (\forall s.\ \ P\ s < \infty \longrightarrow (\exists t\ p.\ ((c,s) \Rightarrow\hat{\ } p\ t) \wedge P\ s \geq p + Q\ t))$"

Finally, we define quantitative Hoare judgements. The idea is that both pre- and post-condition assign an *enat* to a state that is then decreased as the execution progresses. We will see an example in the next step.

inductive *hoareQ* :: *"qassn ⇒ com ⇒ qassn ⇒ bool"* ("$\vdash_Q$ ({(1_)}/ (_)/ {(1_)})" *50*) **where**

— Skipping and assignment both decrease the potential.
SkipQ: "$\vdash_Q$ {λs. *eSuc* (*P s*)} *SKIP* {*P*}" |
AssignQ: "$\vdash_Q$ {λs. *eSuc* (*P* (*s*[*a*/*x*]))} *x*::=*a* {*P*}" |

— *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 ∞ 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 *False* as precondition.
IfQ: "⟦ $\vdash_Q$ {λs. *P s* + ↓(*bval b s*)} c_1 {*Q*};
$\vdash_Q$ {λs. *P s* + ↓(¬ *bval b s*)} c_2 {*Q*} ⟧
⟹ $\vdash_Q$ {λs. *eSuc* (*P s*)} *IF b THEN* c_1 *ELSE* c_2 {*Q*}" |

— Sequence works about as expected.
SeqQ: "⟦ $\vdash_Q$ {P_1} c_1 {P_2}; $\vdash_Q$ {P_2} c_2 {P_3}⟧ ⟹ $\vdash_Q$ {P_1} c_1;;c_2 {P_3}" |

— *WHILE _ DO _* is a combination of conditional and sequence. The invariant is also a function to *enat*.
WhileQ:
"$\vdash_Q$ {λs. *I s* + ↓(*bval b s*)} *c* {λt. *I t* + *1*}
⟹ $\vdash_Q$ {λs. *I s* + *1*} *WHILE b DO c* {λs. *I s* + ↓(¬ *bval b s*)}" |

— The consequence rule also works like in the classic Hoare calculus.
conseqQ: "⟦ $\vdash_Q$ {*P*} *c* {*Q*}; ⋀s. *P s* ≤ *P' s*; ⋀s. *Q' s* ≤ *Q s* ⟧ ⟹
$\vdash_Q$ {*P'*} *c* {*Q'*}"

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.

definition *wsum* :: *com* **where**
"*wsum* =
''*y*'' ::= *N 0*;;
WHILE Less (*N 0*) (*V* ''*x*'')
DO (''*y*'' ::= *Plus* (*V* ''*y*'') (*V* ''*x*'');;
''*x*'' ::= *Plus* (*V* ''*x*'') (*N* (− *1*)))"

The following lemma states the the *wsum* program will take at most *2* + *3* ∗ *n* steps to complete. Prove it!

theorem *wsum*: "$\vdash_Q$ {λs. *enat* (*2* + *3*∗*n*) + ↓ (*s* ''*x*'' = *int n*)} *wsum* {λs. *0*}"
unfolding *wsum_def*
apply(*rule SeqQ*[*rotated*])

apply(*rule conseqQ*)
  apply(*rule WhileQ*[**where** I= "$\lambda s.$ *enat* (*3* $*$ *nat* (*s* ''*x*''))"])

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*: "$\models_Q$ {$\lambda a.$ *eSuc* (*P a*)} *SKIP* {*P*}"
unfolding *hoare_Qvalid_def* **proof** (*safe*)
  fix *s* **assume** "*eSuc* (*P s*) < ∞"
  then have "(*SKIP*, *s*) $\Rightarrow$^*1 s* $\wedge$ *enat 1* + *P s* $\leq$ *eSuc* (*P s*)"
    using *Skip eSuc_def* **by** (*auto split*: *enat.splits*)
  thus "$\exists$ *t n*. (*SKIP*, *s*) $\Rightarrow$^*n t* $\wedge$ *enat n* + *P t* $\leq$ *eSuc* (*P s*)"
    by *blast*
qed

theorem *Assign_sound*: "$\models_Q$ {$\lambda b.$ *eSuc* (*P* (*b*[*a*/*x*]))} *x*::=*a* {*P*}"
theorem *conseq_sound*:
  assumes *hyps*: "$\bigwedge s.$ *P s* $\leq$ *P' s*" "$\bigwedge s.$ *Q' s* $\leq$ *Q s*"
  assumes *IH*: "$\models_Q$ {*P*} *c* {*Q*}"
  shows "$\models_Q$ {*P'*} *c* {*Q'*}"

theorem *If_sound*:
  assumes "$\models_Q$ {$\lambda a.$ *P a* + $\downarrow$ (*bval b a*)} c_1 {*Q*}"
  assumes " $\models_Q$ {$\lambda a.$ *P a* + $\downarrow$ ($\neg$ *bval b a*)} c_2 {*Q*}"
  shows "$\models_Q$ {$\lambda a.$ *eSuc* (*P a*)} *IF b THEN* c_1 *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*: "$\vdash_Q$ {*P*} *c* {*Q*} $\Longrightarrow$ $\models_Q$ {*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:

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 bval 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*