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

WS 23/24
9.1.2024

Semantics of Programming Languages

Exercise Sheet 10

Exercise 10.1 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 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!

Homework 10.1 A Hoare Calculus with Execution Times

Submission until Monday, Jan 15, 23:59pm.

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 $(c, s) \Rightarrow^n t$ 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:

$(SKIP, s) \Rightarrow^1 s$

$(x ::= a, s) \Rightarrow^1 s[a/x]$

$[\![(c_1, s_1) \Rightarrow^{n_1} s_2; (c_2, s_2) \Rightarrow^{n_2} s_3; n_1 + n_2 = n_3]\!] \Longrightarrow (c_1;; c_2, s_1) \Rightarrow^{n_3} s_3$

$[\![bval\ b\ s; (c_1, s) \Rightarrow^{n_1} t; n_3 = Suc\ n_1]\!] \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \Rightarrow^{n_3} t$

$[\![\neg\ bval\ b\ s; (c_2, s) \Rightarrow^{n_2} t; n_3 = Suc\ n_2]\!] \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \Rightarrow^{n_3} t$

$\neg\ bval\ b\ s \Longrightarrow (WHILE\ b\ DO\ c, s) \Rightarrow^1 s$

$[\![bval\ b\ s_1; (c, s_1) \Rightarrow^{n_1} s_2; (WHILE\ b\ DO\ c, s_2) \Rightarrow^{n_2} s_3; n_1 + n_2 + 1 = n_3]\!] \Longrightarrow$
$(WHILE\ b\ DO\ c, s_1) \Rightarrow^{n_3} s_3$

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:

↓ *False* = ∞
↓ *True* = *0*

We can define what it means for a quantitative Hoare triple to be valid:
$(\models_Q \{P\}\ c\ \{Q\}) = (\forall s.\ P\ s < \infty \longrightarrow (\exists t\ p.\ (c,\ s) \Rightarrow\hat{}p\ t \wedge enat\ p + Q\ t \leq P\ s))$
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"* (*"⊢Q ({(1_)}/ (_)/ {(1_)})" 50*) **where**

— Skipping and assignment both decrease the potential.
SkipQ: *"⊢Q {λs. eSuc (P s)} SKIP {P}"* |
AssignQ: *"⊢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: *"⟦ ⊢Q {λs. P s + ↓(bval b s)} c1 {Q};*
⊢Q {λs. P s + ↓(¬ bval b s)} c2 {Q} ⟧
⟹ ⊢Q {λs. eSuc (P s)} IF b THEN c1 ELSE c2 {Q}" |

— Sequence works about as expected.
SeqQ: *"⟦ ⊢Q {P1} c1 {P2}; ⊢Q {P2} c2 {P3}⟧ ⟹ ⊢Q {P1} c1;;c2 {P3}"* |

— *WHILE _ DO _* is a combination of conditional and sequence. The invariant is also a function to *enat*.
WhileQ:
"⊢Q {λs. I s + ↓(bval b s)} c {λt. I t + 1}
⟹ ⊢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*: "$\vdash_Q$ {λ*s*. *enat* (*2* + *3*∗*n*) + ↓ (*s ''x'' = int n*)} *wsum* {λ*s*. *0*}"
unfolding *wsum_def*
apply(*rule SeqQ*[*rotated*])
apply(*rule conseqQ*)
apply(*rule WhileQ*[**where** *I*="λ*s*. *enat* (*3* ∗ *nat* (*s ''x''*))"])

Step 5 You task is to prove soundness. The SKIP-case is already demonstrated below. Prove the remaining extracted lemmas.

lemma *Skip_sound*: "$\models_Q$ {λ*a*. *eSuc* (*P a*)} *SKIP* {*P*}"
unfolding *hoare_Qvalid_def* **proof** (*safe*)
fix *s* **assume** "*eSuc* (*P s*) < ∞"
then have "(*SKIP*, *s*) ⇒^*1 s* ∧ *enat 1* + *P s* ≤ *eSuc* (*P s*)"
using *Skip eSuc_def* **by** (*auto split*: *enat.splits*)
thus "∃ *t n*. (*SKIP*, *s*) ⇒^*n t* ∧ *enat n* + *P t* ≤ *eSuc* (*P s*)"
by *blast*
qed

theorem *Assign_sound*: "$\models_Q$ {λ*b*. *eSuc* (*P* (*b*[*a*/*x*]))} *x*::=*a* {*P*}"
theorem *conseq_sound*:
assumes *hypP*: "⋀*s*. *P s* ≤ *P′ s*"
and *hypQ*: "⋀*s*. *Q′ s* ≤ *Q s*"
and *IH*: "$\models_Q$ {*P*} *c* {*Q*}"
shows "$\models_Q$ {*P′*} *c* {*Q′*}"

theorem *If_sound*:
assumes "$\models_Q$ {λ*a*. *P a* + ↓ (*bval b a*)} c_1 {*Q*}"
and " $\models_Q$ {λ*a*. *P a* + ↓ (¬ *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 $\{P_2\}$"
and "$\models_Q$ $\{P_2\}$ c_2 $\{P_3\}$"
shows "$\models_Q$ $\{P_1\}$ $c_1;;c_2$ $\{P_3\}$"

Hint: You'll need to induct over the potential in the last proof – for this, the *less_induct* induction scheme may be helpful.

theorem *While_sound*:
assumes "$\models_Q$ $\{\lambda s.\ INV\ s + \downarrow(bval\ b\ s)\}$ c $\{\lambda t.\ INV\ t + 1\}$"
shows "$\models_Q$ $\{\lambda s.\ INV\ s + 1\}$ *WHILE* b *DO* c $\{\lambda s.\ INV\ s + \downarrow(\neg\ bval\ b\ s)\}$"