Final Exam

Semantics of Programming Languages
Solution

22. 2. 2024

First name: ______________________

Last name: ______________________

Student-Id (Matrikelnummer): ______________________

Signature: ______________________

1. The exam is to be solved in Isabelle on your own Laptop.
2. Submit your solutions to do.proof.in.tum.de, in the **Semantics Exam 2023/2024** contest. You may submit any number of times, only your last submission will be counted.
3. You may use all lecture material (including exercises and your homework) to solve the exam.
4. Using the internet is not allowed for any other reason than submitting your solution to the submission system.
5. You have 120 minutes to solve the exam and submit your solutions.
6. Please put your student ID and ID-card or driver's license on the table until we have checked it.
7. You may not leave the room in the last 15 minutes of the exam — you may disturb other students who need this time.

Proof Guidelines: We expect valid Isabelle proofs. `sledgehammer` may be used. The use of `sorry` may lead to the deduction of points but is preferable to spending a lot of time on individual proof steps. Unfinished proofs should be well written and easy to understand!

1 Induction (solution)

```
inductive to_zero :: "nat list ⇒ bool" where
tz0: "to_zero [0]" |
tzS: "to_zero (x # xs) ⟹ to_zero (Suc x # x # xs)"

fun enum :: "nat ⇒ nat list" where
"enum 0 = [0]" |
"enum (Suc n) = Suc n # enum n"

lemma enum_if_to_zero:
  assumes "to_zero xs"
  shows "∃ n. enum n = xs"
using assms
proof (induction xs rule: to_zero.induct)
  case tz0
  then show ?case using enum.simps by blast
next
  case (tzS x xs) then show ?case by (metis enum.cases enum.simps list.inject)
qed

lemma to_zero_enum: "to_zero (enum n)"
proof (induction n)
  case (Suc n)
  then show ?case
    by (auto intro: to_zero.intros split: if_splits) (metis to_zero.simps enum.elims)
qed (auto intro: to_zero.intros)
```

2 IMP (solution)

```
inductive big_step :: "com × state ⇒ state ⇒ bool" (infix "⇒" 55) where
Skip: "(SKIP,s) ⇒ s" |
Assign: "(x ::= a,s) ⇒ s(x := aval a s)" |
Seq: "⟦ (c1,s1) ⇒ s2;  (c2,s2) ⇒ s3 ⟧ ⟹ (c1;;c2, s1) ⇒ s3" |
IfTrue: "⟦ bval b s;  (c1,s) ⇒ t ⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t" |
IfFalse: "⟦ ¬bval b s;  (c2,s) ⇒ t ⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t" |
LoopFalse: "n = 0 ⟹ (LOOP n DO c,s) ⇒ s" |
LoopTrue: "⟦0 < n; (c,s1) ⇒ s2;  (LOOP (n − 1) DO c, s2) ⇒ s3 ⟧ ⟹ (LOOP n DO c, s1)
⇒ s3"

declare big_step.intros [intro]
inductive_cases SkipE[elim!]: "(SKIP,s) ⇒ t"
inductive_cases AssignE[elim!]: "(x ::= a,s) ⇒ t"
inductive_cases SeqE[elim!]: "(c1;;c2,s1) ⇒ s3"
inductive_cases IfE[elim!]: "(IF b THEN c1 ELSE c2,s) ⇒ t"
inductive_cases LoopE[elim]: "(LOOP n DO c,s) ⇒ t"

definition "gt_zero n ≡ less (N 0) (N (int n))"

lemma bval_gt_zero [iff]: "bval (gt_zero n) s ⟷ 0 < n"
  unfolding gt_zero_def by auto

inductive small_step :: "com ∗ state ⇒ com ∗ state ⇒ bool" (infix "→" 55) where
Assign:  "(x ::= a, s) → (SKIP, s(x := aval a s))" |
Seq1:    "(SKIP;;c2,s) → (c2,s)" |
Seq2:    "(c1,s) → (c1',s') ⟹ (c1;;c2,s) → (c1';;c2,s')" |
IfTrue:  "bval b s ⟹ (IF b THEN c1 ELSE c2,s) → (c1,s)" |
IfFalse: "¬bval b s ⟹ (IF b THEN c1 ELSE c2,s) → (c2,s)" |
Loop: "(LOOP n DO c,s) → (IF gt_zero n THEN c;; LOOP (n − 1) DO c ELSE SKIP,s)"

inductive_cases sSkipE[elim!]: "(SKIP,s) → ct"
inductive_cases sAssignE[elim!]: "(x::=a,s) → ct"
inductive_cases sSeqE[elim]: "(c1;;c2,s) → ct"
inductive_cases sIfE[elim!]: "(IF b THEN c1 ELSE c2,s) → ct"
inductive_cases sLoopE[elim]: "(LOOP n DO c, s) → ct"

lemma big_to_small: "cs ⇒ t ⟹ cs →∗ (SKIP,t)"
  by (induction cs t rule: big_step.induct) fastforce+

lemma small1_big_continue: "cs → cs' ⟹ cs' ⇒ t ⟹ cs ⇒ t"
  by (induction arbitrary: t rule: small_step.induct) auto

lemma small_to_big: "cs →∗ (SKIP,t) ⟹ cs ⇒ t"
  by (induction cs "(SKIP,t)" rule: star.induct)
  (auto intro: small1_big_continue)
```

```
lemma terminates: "∃ t. (c, s) ⇒ t"
proof (induction c arbitrary: s)
  case (Loop n c)
  then show ?case
  proof (induction n arbitrary: s)
    case (Suc n)
    then show ?case by (metis LoopTrue diff_Suc_1 zero_less_Suc)
  qed auto
qed blast+
```

3 Hoare Logic (solution)

```
inductive
  hoare :: "assn ⇒ com ⇒ assn ⇒ bool" (‹⊢ ({(1_)}/ (_)/ {(1_)})› 50)
where
Skip: "⊢ {P} SKIP {P}" |
Assign: "⊢ {λs. P(s[a/x])} x::=a {P}" |
Seq: "⟦ ⊢ {P} c₁ {Q}; ⊢ {Q} c₂ {R} ⟧ ⟹ ⊢ {P} c₁;;c₂ {R}" |
If: "⟦ ⊢ {λs. P s ∧ bval b s} c₁ {Q}; ⊢ {λs. P s ∧ ¬ bval b s} c₂ {Q} ⟧
    ⟹ ⊢ {P} IF b THEN c₁ ELSE c₂ {Q}" |
While: "⊢ {λs. P s ∧ bval b s} c {P} ⟹
        ⊢ {P} WHILE b DO c {λs. P s ∧ ¬ bval b s}" |
conseq: "⟦ ∀s. P' s ⟶ P s; ⊢ {P} c {Q}; ∀s. Q s ⟶ Q' s ⟧
        ⟹ ⊢ {P'} c {Q'}" |
Or: "⟦ ⊢ {P} c₁ {Q}; ⊢ {P} c₂ {Q} ⟧ ⟹ ⊢ {P} c₁ OR c₂ {Q}" |
Assume: "⊢ {P} ASSUME b {λs. P s ∧ bval b s}"


definition wp_Or :: "com ⇒ com ⇒ assn ⇒ assn" where
  "wp_Or c1 c2 Q s ≡ wp c1 Q s ∧ wp c2 Q s"

definition wp_Assume :: "bexp ⇒ assn ⇒ assn" where
  "wp_Assume b Q s ≡ if bval b s then Q s else True"

lemma wp_Or: "wp (c1 OR c2) Q s = wp_Or c1 c2 Q s"
  unfolding wp_def by (auto simp: wp_def wp_Or_def)

lemma wp_Assume: "wp (ASSUME b) Q s = wp_Assume b Q s"
  by (auto simp: wp_def wp_Assume_def)

lemma wp_is_pre: "⊢ {wp c Q} c {Q}"
proof(induction c arbitrary: Q)
  case If thus ?case by(auto intro: conseq)
next
  case (While b c)
  let ?w = "WHILE b DO c"
  show "⊢ {wp ?w Q} ?w {Q}"
  proof(rule While')
    show "⊢ {λs. wp ?w Q s ∧ bval b s} c {wp ?w Q}"
    proof(rule strengthen_pre[OF _ While.IH])
      show "∀s. wp ?w Q s ∧ bval b s ⟶ wp c (wp ?w Q) s" by auto
    qed
    show "∀s. wp ?w Q s ∧ ¬ bval b s ⟶ Q s" by auto
  qed
next
  case (Assume x)
  then show ?case
    by (metis (no_types, lifting) hoare.Assume weaken_post wp_Assume wp_Assume_def)
```

next
  case (*Or c1 c2*)
  then show *?case*
    by (*metis hoare.Or strengthen_pre wp_Or wp_Or_def*)
qed *auto*

lemma *hoare_complete*: **assumes** "$\models \{P\}c\{Q\}$" **shows** "$\vdash \{P\}c\{Q\}$"
proof(*rule strengthen_pre*)
  show "$\forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$" **using** *assms*
    by (*auto simp*: *hoare_valid_def wp_def*)
  show "$\vdash \{wp\ c\ Q\}\ c\ \{Q\}$" **by**(*rule wp_is_pre*)
qed

corollary *hoare_sound_complete*: "$\vdash \{P\}c\{Q\} \longleftrightarrow \models \{P\}c\{Q\}$"
by (*metis hoare_complete hoare_sound*)

4 Abstract Listification (solution)

```
fun square_eint :: "eint ⇒ eint" where
"square_eint NInf = NInf" |
"square_eint PInf = PInf" |
"square_eint (Z i) = PInf"

fun append_eint :: "eint ⇒ eint ⇒ eint" where
"append_eint PInf _ = PInf" |
"append_eint _ PInf = PInf" |
"append_eint NInf xs = xs" |
"append_eint xs NInf = xs" |
"append_eint (Z i) (Z j) = Z (max i j)"

fun take_eint :: "nat ⇒ eint ⇒ eint" where
"take_eint 0 _ = NInf" |
"take_eint _ ei = ei"

lemma square_sound: "⟦ xs ∈ γ_eint ei ⟧ ⟹ square xs ∈ γ_eint (square_eint ei)"
by (cases ei) (auto simp: square_def)

lemma append_sound:
  "⟦ xs1 ∈ γ_eint ei1; xs2 ∈ γ_eint ei2 ⟧ ⟹ (xs1 @ xs2) ∈ γ_eint (append_eint ei1 ei2)"
by (cases ei1; cases ei2; cases xs1; cases xs2) auto

lemma take_sound: "⟦ xs ∈ γ_eint ei ⟧ ⟹ take n xs ∈ γ_eint (take_eint n ei)"
apply (cases ei; cases n)
apply auto
using not_less_eq set_take_subset by fastforce


fun inv_square :: "eint ⇒ eint ⇒ eint" where "inv_square _ _ = undefined"

definition inv_square_correct :: bool where
  "inv_square_correct ≡ ∀ e e1 e1' xs.
    inv_square e e1 = e1' ∧ xs ∈ γ_eint e1 ∧ square xs ∈ γ_eint e ⟶ xs ∈ γ_eint e1'"
```