Final Exam

Semantics of Programming Languages

27. 2. 2025

First name: _____

Last name: _____

Student-Id (Matrikelnummer): _____

Signature: _____

1. The exam has to be solved in Isabelle/HOL on your own laptop.
2. Submit your solutions to do.proof.in.tum.de, in the **Semantics Exam 2024/2025** 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.
7. You may not leave the room in the last 15 minutes of the exam because it could 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

Consider the following inductive predicate:

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

Define an executable, recursive function *enum* :: *nat ⇒ nat list* that makes the following two theorems true and prove the theorems!

fun *enum* :: *"nat ⇒ nat list"* **where**

Now prove:

lemma *enum_if_to_zero*:
  assumes *"to_zero xs"*
  shows *"∃ n. enum n = xs"*

lemma *to_zero_enum*: *"to_zero (enum n)"*

2 IMP

Consider a modification of IMP with loops of the form "*LOOP n DO c*", where c is an IMP program and n a natural number denoting the number of times the program c should be executed.

datatype *com* = *SKIP*
| *Assign vname aexp* ("_ ::= _" [1000, 61] 61)
| *Seq com com* ("_;;/ _" [60, 61] 60)
| *If bexp com com* ("(IF _/ THEN _/ ELSE _)" [0, 0, 61] 61)
| *Loop nat com* ("(LOOP _/ DO _)" [0, 61] 61)

Complete the big-step semantics for this language by providing the missing rules for the loop constructor. Do not modify the pre-defined big-step rules!

inductive *big_step* :: "*com* × *state* ⇒ *state* ⇒ *bool*" (**infix** "⇒" 55) **where**
Skip: "(*SKIP*,*s*) ⇒ *s*" |
Assign: "(*x* ::= *a*,*s*) ⇒ *s*(*x* := *aval a s*)" |
Seq: "⟦ (*c*$_1$,*s*$_1$) ⇒ *s*$_2$; (*c*$_2$,*s*$_2$) ⇒ *s*$_3$ ⟧ ⟹ (*c*$_1$;;*c*$_2$, *s*$_1$) ⇒ *s*$_3$" |
IfTrue: "⟦ *bval b s*; (*c*$_1$,*s*) ⇒ *t* ⟧ ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t*" |
IfFalse: "⟦ ¬*bval b s*; (*c*$_2$,*s*) ⇒ *t* ⟧ ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$, *s*) ⇒ *t*" |

Now complete the small-step semantics by providing *one single missing rule* for the loop constructor. Do not modify the pre-defined small-step rules and do not use more than one rule! Your new rule must:

- be properly small-step; that is, do not re-use the big-step semantics, only do a single step, etc.
- not use Isabelle/HOL case analyses. In particular, do not use "*if b then x else y*" and "*case x of ...*"; but you may very well use the other IMP constructors.

inductive *small_step* :: "*com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool*" (**infix** "→" 55) **where**
Assign: "(*x* ::= *a*, *s*) → (*SKIP*, *s*(*x* := *aval a s*))" |
Seq1: "(*SKIP*;;*c*$_2$,*s*) → (*c*$_2$,*s*)" |
Seq2: "(*c*$_1$,*s*) → (*c*$_1$′,*s*′) ⟹ (*c*$_1$;;*c*$_2$,*s*) → (*c*$_1$′;;*c*$_2$,*s*′)" |
IfTrue: "*bval b s* ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$,*s*) → (*c*$_1$,*s*)" |
IfFalse: "¬*bval b s* ⟹ (*IF b THEN c*$_1$ *ELSE c*$_2$,*s*) → (*c*$_2$,*s*)" |

Next show the equivalence between your big-step and small-step semantics by filling out the missing proof steps in the Isabelle theory file.

And finally, prove that all programs terminate:

lemma *terminates*: "∃ *t*. (*c*, *s*) ⇒ *t*"

3 Hoare Logic

Consider the modification of IMP with nondeterministic choice "c_1 *OR* c_2" and the assumption command "*ASSUME b*" as known from the tutorial, week 5; that is "c_1 *OR* c_2" decides nondeterministically to execute c_1 or c_2; and "*ASSUME b*" behaves like *SKIP* if b evaluates to true and returns no result otherwise.

First, complete the Hoare calculus rules for partial correctness of this language by providing the missing rules for the new constructors. Do not modify the pre-defined rules!

inductive
hoare :: *"assn ⇒ com ⇒ assn ⇒ bool"* (‹⊢ ({(*1*_)}/ (_)/ {(*1*_)})› *50*)
where
Skip: "⊢ {*P*} *SKIP* {*P*}" |
Assign: "⊢ {λ*s*. *P*(*s*[*a*/*x*])} *x*::=*a* {*P*}" |
Seq: "⟦ ⊢ {*P*} c_1 {*Q*}; ⊢ {*Q*} c_2 {*R*} ⟧ ⟹ ⊢ {*P*} c_1;;c_2 {*R*}" |
If: "⟦ ⊢ {λ*s*. *P s* ∧ *bval b s*} c_1 {*Q*}; ⊢ {λ*s*. *P s* ∧ ¬ *bval b s*} c_2 {*Q*} ⟧
⟹ ⊢ {*P*} *IF b THEN* c_1 *ELSE* c_2 {*Q*}" |
While: "⊢ {λ*s*. *P s* ∧ *bval b s*} *c* {*P*} ⟹
⊢ {*P*} *WHILE b DO c* {λ*s*. *P s* ∧ ¬ *bval b s*}" |
conseq: "⟦ ∀ *s*. *P*′ *s* ⟶ *P s*; ⊢ {*P*} *c* {*Q*}; ∀ *s*. *Q s* ⟶ *Q*′ *s* ⟧
⟹ ⊢ {*P*′} *c* {*Q*′}" |

Next give appropriate, simple definitions *wp_Or* and *wp_Assume* for the weakest preconditions of "*c1 OR c2*" and "*ASSUME b*", respectively. The definition must subject to the following conditions:

- It holds that "*wp* (*c1 OR c2*) *Q* = *wp_Or c1 c2 Q*" and "*wp* (*ASSUME b*) *Q* = *wp_Assume b Q*".
- The rules must be simple and not merely be a restatement of the definition of *wp*. In particular, your definitions may not use quantifiers (forall, exists).
- The definition of "*wp_Or c1 c2 Q*" for "*c1 OR c2*" must use "*wp c1 Q*" as well as "*wp c2 Q*".

definition *wp_Or* :: *"com ⇒ com ⇒ assn ⇒ assn"* **where**
definition *wp_Assume* :: *"bexp ⇒ assn ⇒ assn"* **where**

Finally show that your Hoare rules are sound and complete by filling out the missing proof steps in the Isabelle theory file.

4 Abstract Listification

For this exercise, we will consider a modification of IMP that computes on integer lists:

type_synonym *vname = string*
type_synonym *val = "int list"*
type_synonym *state = "vname ⇒ val"*

Next to constants and variables, the language offers expressions to

- square all numbers contained in a list,
- append two lists, and
- take n elements of a list.

datatype *exp = N val | V vname | Square exp | Append exp exp | Take nat exp*

The semantics is as follows:

definition *square :: "val ⇒ val"* **where** *"square ≡ map (λx. x * x)"*

fun *lval :: "exp ⇒ state ⇒ val"* **where**
"lval (N xs) s = xs" |
"lval (V x) s = s x" |
"lval (Square e) s = square (lval e s)" |
"lval (Append e1 e2) s = lval e1 s @ lval e2 s" |
"lval (Take n e) s = take n (lval e s)"

Consider an abstract interpreter on the domain consisting of $\mathbb{Z} \cup \{-\infty, \infty\}$ for this language, indicating the least upper bound of all elements contained in a list:

datatype *eint = Z int*
| *NInf* — negative infinity "$-\infty$"
| *PInf* — positive infinity "∞"

The concretization and abstraction functions are as follows:

fun *γ_eint :: "eint ⇒ val set"* **where**
"γ_eint NInf = {[]}" |
"γ_eint PInf = UNIV" |
"γ_eint (Z i) = {xs. xs = [] ∨ Max (set xs) ≤ i}"

fun *num_eint :: "val ⇒ eint"* **where**
"num_eint [] = NInf" |
"num_eint xs = Z (Max (set xs))"

Your task is to define the abstract operations and to show that they are sound with respect to the concretization function. Your abstract operations should be as precise as possible!

Hint for the proofs: If you want to perform nested case splittings on variables *x_1*,...,*x_n*, you can use *apply (cases x_1; cases x_2; ...; cases x_n)*.

fun *square_eint :: "eint ⇒ eint"*

fun *append_eint* :: *"eint ⇒ eint ⇒ eint"*

fun *take_eint* :: *"nat ⇒ eint ⇒ eint"*

lemma *square_sound*: *"⟦ xs ∈ γ_eint ei ⟧ ⟹ square xs ∈ γ_eint (square_eint ei)"*

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

lemma *take_sound*: *"⟦ xs ∈ γ_eint ei ⟧ ⟹ take n xs ∈ γ_eint (take_eint n ei)"*

Now assume that we have an inverse analyses *inv_square* :: *eint ⇒ eint ⇒ eint* for *Square*. Write down the statement of the correctness theorem for *inv_square* as a definition called *inv_square_correct*:

definition *inv_square_correct* :: *bool* **where**