Logical Verification 2018–2019
Vrije Universiteit Amsterdam
Lecturers: dr. J. C. Blanchette and dr. J. Hölzl

Repeat Exam
Tuesday 5 February 2019, 18:30–21:15, WN-S655
5 questions, 90 points
Answers may be given in English or Dutch

Proof Guidelines

We expect detailed, rigorous, mathematical proofs, but we do not ask you to write Lean proofs. You may use standard mathematical notation or Lean structured commands (e.g., `assume`, `have`, `show`, `calc`). You may also use tactical proofs (e.g., `intro`, `apply`), but then please indicate some of the intermediate goals, so that we can follow the chain of reasoning.

Major proof steps, especially applications of induction and invocation of the induction hypothesis, must be stated explicitly. For each case of a proof by induction, you must list the inductive hypotheses assumed (if any) and the goal to be proved. Minor proof steps corresponding to `refl`, `simp`, or arithmetic need not be justified if you think they are obvious (to humans), but you should say which key lemmas they follow from.

You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate, especially for functions and predicates that are specific to an exam question.

In Case of Ambiguities or Errors in an Exam Question

The staff present at the exam has the lecturers' phone numbers, in case of questions or issues concerning a specific exam question. Nevertheless, we strongly recommend that you work things out yourselves, stating explicitly any ambiguity or error and explaining how you interpret or repair the question. The more explicit you are, the easier it will be for the lecturers to grade the question afterwards.

Question 1. WHILE language with print (6+6+6 points)

Consider the following Lean definition of the syntax of the "PRINT language," a variant of the familiar WHILE language featuring a `print` statement, which sends a string to the standard output channel:

```
inductive pl (σ : Type) : Type
| skip  {} : pl
| print {} : string → pl
| assign   : (σ → σ) → pl
| seq      : pl → pl → pl
| ite      : (σ → Prop) → pl → pl → pl
| while    : (σ → Prop) → pl → pl
```

1a. The big-step semantics relation $\Longrightarrow$ relates pairs (p, s), where p is a program to execute and s an initial variable state, with pairs (m, t), where m is the output message and t is the final state. Intuitively, $(p, s) \Longrightarrow (m, t)$ means that "starting in a state s, executing p ends in the state t, printing m in the process." For example:

$$\texttt{(seq (print "ab") (print "ba"), s)} \Longrightarrow \texttt{("abba", s)}$$

Give the big-step semantics rules (in traditional rule format or Lean syntax) covering the statements `skip`, `print`, and `seq`.

1b. The small-step semantics relation $\longrightarrow$ relates tuples $(p, (m, s))$, where p is the program to execute, m is the initial contents of the output channel, and s is the initial variable state, with tuples $(p', (m', s'))$, where m' is m extended with any output produced while executing one step of p. For example:

```
    (seq (print "cada") (print "bra"), ("abra", s))
⟶ (seq skip (print "bra"), ("abracada", s))
⟶ (print "bra", ("abracada", s))
⟶ (skip, ("abracadabra", s))
```

Give the small-step semantics rules (in traditional rule format or Lean syntax) covering the statements `skip`, `print`, and `seq`.

1c. Program equivalence $\approx$ is defined in terms of the big-step semantics in the usual way:

$$\texttt{p}_1 \approx \texttt{p}_2 \text{ if and only if } \forall \texttt{s mt, (p}_1\texttt{, s)} \Longrightarrow \texttt{mt} \leftrightarrow \texttt{(p}_2\texttt{, s)} \Longrightarrow \texttt{mt}.$$

Prove or disprove the equivalences below. In the "prove" case, give an informal argument for both directions of the equivalence. In the "disprove" case, give a counterexample, including instantiations of the variables occurring in the equivalences and in the definition of $\approx$, and explain why it is a counterexample.

1. `while b do skip` $\approx$ `skip`
2. `seq (seq p q) r` $\approx$ `seq p (seq q r)`
3. `print m` $\approx$ `skip`

Question 2. 2–3 trees (4+6+4+7 points)

Consider the following Lean definition of 2–3 trees as an inductive type:

```
inductive tree (α : Type) : Type
| empty {} : tree
| bin      : α → tree → tree → tree
| ter      : α → tree → tree → tree → tree
```

2a. Complete the following Lean definition. The `map_tree` function should apply its argument `f` to all values of type α stored in the tree and otherwise preserve the tree's structure.

```
def map_tree {α β : Type} (f : α → β) : tree α → tree β
```

2b. Prove the following lemma about your definition of `map_tree`.

```
lemma map_tree_id {α : Type} :
  ∀t : tree α, map_tree (λx : α, x) t = t
```

2c. Complete the following Lean definition. The `set_tree` function should return the set of all values of type α stored in the tree. In your answer, you may use traditional set notations regardless of whether they are actually supported by Lean.

```
def set_tree {α : Type} : tree α → set α
```

2d. A *congruence rule* is a lemma that can be used to lift an equivalence relation between terms to the same terms occurring under a common context. Congruence rules for equality are built into Lean's logic. In the following example, the equivalence relation is `=`, the terms are `f` and `g`, and the context is `map_tree ... t`:

```
lemma map_tree_congr_weak {α β : Type} (f g : α → β) (f = g) (t : tree α) :
  map_tree f t = map_tree g t :=
by simp *
```

Regrettably, the above rule is not as flexible as it could be. As long as `f` and `g` are equal for all values `x : α` stored in `t`, we have `map_tree f t = map_tree g t`, even if `f` and `g` disagree on other α values. Inspired by this observation, prove the following stronger congruence rule.

```
lemma map_tree_congr_strong {α β : Type} (f g : α → β) :
  ∀t : tree α, (∀x, x ∈ set_tree t → f x = g x) → map_tree f t = map_tree g t
```

Question 3. Hoare logic gone wild (4+4+8 points)

Consider the standard WHILE language, as described in the lecture, and the following definition of partial-correctness Hoare triples:

$$\texttt{\{* P *\} p \{* Q *\}} \text{ if and only if } \forall \texttt{s t, P s} \to \texttt{(p, s)} \implies \texttt{t} \to \texttt{Q t}.$$

3a. Louis Reasoner proposes the following Hoare rule for reasoning about `while` loops:

```
lemma while_intro_lr (h : {* P *} p {* P *}) :
  {* P *} while c p {* λs, P s ∧ ¬ c s *}
```

Is this rule sound? Is it complete? Justify your answer to each question with a brief proof sketch or a counterexample.

3b. Alyssa P. Hacker and Ben Bitdiddle cannot agree on what a sound and complete consequence rule for Hoare logic should look like. Ms. Hacker believes the correct rule is as follows:

```
lemma consequence_aph (h : {* P *} p {* Q *})
    (hp : ∀s, P' s → P s) (hq : ∀s, Q s → Q' s) :
  {* P' *} p {* Q' *}
```

Mr. Bitdiddle is convinced that she has exchanged the premise `h` and the conclusion. He claims the correct rule is

```
lemma consequence_bb (h : {* P' *} p {* Q' *})
    (hp : ∀s, P' s → P s) (hq : ∀s, Q s → Q' s) :
  {* P *} p {* Q *}
```

Who is right? Why?

3c. The standard Hoare rule for reasoning about `if–then–else` statements is as follows:

```
lemma ite_intro
    (h₁ : {* λs, P s ∧   c s *} p₁ {* Q *})
    (h₂ : {* λs, P s ∧ ¬ c s *} p₂ {* Q *}) :
  {* P *} ite c p₁ p₂ {* Q *}
```

Eva Lu Ator has come up with an alternative rule, where the premises' preconditions are variables:

```
lemma ite_intro_ela
    (h₁ : {* R₁ *} p₁ {* S *})
    (h₂ : {* R₂ *} p₂ {* S *}) :
  {* λs, (c s ∧ R₁ s) ∨ (¬ c s ∧ R₂ s) *} ite c p₁ p₂ {* S *}
```

Prove that Ms. Lu Ator's rule is equivalent to the standard rule.

Hint: Equivalence can be established by deriving each rule from the other, *without* expanding the definition of Hoare triples. You can exploit basic properties of the logical connectives and the correct Hoare consequence rule from subquestion 3b.

Question 4. Binary trees as a subset type (3+5+3+4+5 points)

Recall the Lean definition of 2–3 trees from Question 2:

```
inductive tree (α : Type) : Type
| empty {} : tree
| bin      : α → tree → tree → tree
| ter      : α → tree → tree → tree → tree
```

4a. A *binary tree* is a 2–3 tree that is built without using the `ter` constructor. Complete the following Lean definition of an inductive predicate that captures this concept.

```
inductive is_binary {α : Type} : tree α → Prop
| empty : is_binary empty
```

4b. Prove that applying the `map_tree` function from subquestion 2a preserves the binary nature of a binary tree. Formally:

```
lemma is_binary_map_tree {α β : Type} {f : α → β} :
  ∀t : tree α, is_binary t → is_binary (map_tree f t)
```

4c. Use Lean's subtype mechanism to define the type `btree` α of binary trees by carving out the binary trees from the 2–3 trees, by filling in the hole (`_`) below.

```
def btree (α : Type) : Type := _
```

4d. Basic operations on binary trees include the empty binary tree and the binary node constructor, which takes an α value and two binary trees and returns a binary tree.

Fill in the two holes (`_`) below to implement these operations.

Parts of the holes correspond to proof terms. For each of these, state the property that must be proved and explain very briefly why it should hold.

```
def empty_btree {α : Type} : btree α := _
def bin_btree {α : Type} (a : α) (l r : btree α) : btree α := _
```

4e. The `map_tree` and `set_tree` operations defined in subquestions 2a and 2c can be used to define the corresponding operators on binary trees.

Fill in the two holes (`_`) below to implement these operations.

Part of the second hole corresponds to a proof term. State the property that must be proved and explain very briefly why it should hold.

```
def set_btree {α : Type} (t : btree α) : set α := _
def map_btree {α β : Type} (f : α → β) (t : btree α) : btree β := _
```

Question 5. Reflexive symmetric closure (4+7+4 points)

In mathematics, the reflexive symmetric closure $\Leftrightarrow^=$ of a binary relation $\Rightarrow$ over a set A is defined as the smallest relation satisfying the following rules:

(base) for all $a, b \in A$, if $a \Rightarrow b$, then $a \Leftrightarrow^= b$;

(refl) for all $a \in A$, $a \Leftrightarrow^= a$;

(symm) for all $a, b \in A$, if $a \Leftrightarrow^= b$, then $b \Leftrightarrow^= a$.

5a. Completing the Lean definition below of a predicate `rsc` whose introduction rules closely follow the rules **(base)**, **(refl)**, and **(symm)**:

```
inductive rsc {α : Type} (r : α → α → Prop) : α → α → Prop
```

5b. Prove that the reflexive symmetric closure of a relation $\Rightarrow$ that is reflexive and symmetric is equivalent to the relation $\Rightarrow$ itself. Formally:

```
lemma refl_symm_imp_rsc_iff {α : Type} {r : α → α → Prop}
    (refl_r : ∀a, r a a) (symm_r : ∀a b, r a b → r b a) :
  ∀a b : α, rsc r a b ↔ r a b
```

Hint: Start with the easy direction. For the other direction, use rule induction.

5c. Use the lemma `refl_symm_imp_rsc_iff` from subquestion 5b to derive `rsc`'s idempotence. Formally:

```
lemma rsc_rsc_iff_rsc {α : Type} {r : α → α → Prop} :
  ∀a b : α, rsc (rsc r) a b ↔ rsc r a b
```

The grade for the exam is the total amount of points divided by 10, plus 1.