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

Final Exam
Tuesday 18 December 2018, 15:15–18:00, WN-Q112
5 problems, 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 are welcome to use standard mathematical notation or Lean structured commands (e.g., `assume`, `have`, `show`, `calc`). You can 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. Transitive closure (3+3+5+7 points)

In mathematics, the transitive closure R^+ of a binary relation R over a set A can be defined as the smallest solution satisfying the following rules:

(base) for all $a, b \in A$, if $a\ R\ b$, then $a\ R^+\ b$;

(step) for all $a, b, c \in A$, if $a\ R\ b$ and $b\ R^+\ c$, then $a\ R^+\ c$.

In Lean, we can define this concept as follows:

```
inductive tc₁ {α : Type} (r : α → α → Prop) : α → α → Prop
| base : ∀a b, r a b → tc₁ a b
| step : ∀a b c, r a b → tc₁ b c → tc₁ a c
```

1a. Rule **(step)** makes it convenient to extend transitive chains by adding links to the left. Another way to define the transitive closure R^+ would use the following rule instead of **(step)**, with a preference for the right:

(pets) for all $a, b, c \in A$, if $a\ R^+\ b$ and $b\ R\ c$, then $a\ R^+\ c$.

Define, using Lean syntax, a predicate `tc₂` that embodies this alternative definition.

1b. Yet another definition of the transitive closure R^+ would use the following symmetric rule instead of **(step)** or **(pets)**:

(trans) for all $a, b, c \in A$, if $a\ R^+\ b$ and $b\ R^+\ c$, then $a\ R^+\ c$.

Define, using Lean syntax, a predicate `tc₃` that embodies this alternative definition.

1c. Prove that **(step)** also holds as a lemma about `tc₃` as defined in subquestion 1b above. In Lean syntax:

```
lemma tc₃_step {α : Type} (r : α → α → Prop) :
  ∀a b c : α, r a b → tc₃ r b c → tc₃ r a c
```

1d. Prove by rule induction on the `tc₁ r a b` premise below that **(pets)** also holds as a lemma about `tc₁` as defined above.

```
lemma tc₁_pets {α : Type} (r : α → α → Prop) :
  ∀a b c : α, tc₁ r a b → r b c → tc₁ r a c
```

Question 2. Element count in a list (5+6+7 points)

(For this question, we assume that we work in a classical logic, in which all propositions are considered decidable and noncomputable functions are allowed. In Lean, this is achieved by the following declarations:

```
local attribute [instance, priority 0] classical.prop_decidable

noncomputable theory )
```

Consider the following Lean definitions on the predefined `list` type:

```
def mem {α : Type} (a : α) : list α → Prop
| []        := false
| (x :: xs) := x = a ∨ mem xs

def reverse {α : Type} : list α → list α
| []        := []
| (x :: xs) := reverse xs ++ [x]
```

2a. Complete the following definition of a function `count a xs` that returns the number of occurrences of a value `a` in a list `xs`.

```
def count {α : Type} (a : α) : list α → ℕ
```

2b. Prove the following lemma, which relates the `count` function defined in subquestion 2a with the `mem` function defined above.

```
lemma count_eq_zero_iff_not_mem {α : Type} (a : α) :
  ∀xs, count a xs = 0 ↔ ¬ mem a xs
```

2c. Prove the following lemma, involving the `reverse` function defined above.

```
lemma count_reverse_eq_count {α : Type} (a : α) :
  ∀xs : list α, count a (reverse xs) = count a xs
```

Make sure to state and prove any nontrivial auxiliary lemmas you may need to reason about the `count` or `reverse` function.

Question 3. Finite multisets as quotients and subsets (4+6+5+6+4 points)

(As for Question 2, for this question, we assume that we work in a classical logic, in which all propositions are considered decidable and noncomputable functions are allowed.)

A multiset (or bag) is a collection of elements that allows for multiple (but finitely many) occurrences of its elements. For example, $\{3,5\} = \{5,3\} \neq \{5,3,3\}$.

Finite multisets can be defined in at least two different ways. We will explore these.

3a. One way to define finite multisets is to start with a raw type of finite lists and to consider only the *number of occurrences* of elements in the lists, ignoring the order in which elements occur. Following this scheme, `[5, 3, 3]`, `[3, 5, 3]`, and `[3, 3, 5]` would be three representations of the multiset $\{5,3,3\}$.

In Lean, this could be achieved as follows:

```
instance multiset₁.rel (α : Type) : setoid (list α) :=
{ r     := _,
  iseqv :=
  ⟨ (assume xs, by simp),
    (assume xs ys ys_xs x, by simp *),
    (assume xs ys zs xs_ys ys_zs, by simp *) ⟩ }

def multiset₁ (α : Type) : Type := quotient (multiset₁.rel α)
```

However, there is a hole (`_`) in the definition, standing for the term specifying the equivalence relation `r` used for the quotient construction.

Supply the missing term. You may use the `count` function from Question 2 if it helps.

3b. Basic operations on multisets include the empty multiset ($\emptyset$), the singleton multiset ($\{a\}$ for any element a), and the sum of two multisets ($A \uplus B$ for any multisets A and B). The sum should be defined so that the multiplicity of elements are added; for example, $\{1\} \uplus \{1,1\} = \{1,1,1\}$.

Fill in the three holes (`_`) below to implement the basic multiset operations.

- For the first two holes, actual Lean terms are expected.
- For the third hole, state the property that must be proved and explain briefly why it holds. You may assume reasonable lemmas about `count`.

```
def empty_mset₁ {α : Type} : multiset₁ α := ⟦[]⟧

def singleton_mset₁ {α : Type} (a : α) : multiset₁ α := _

def sum_mset₁ {α : Type} (A : multiset₁ α) (B : multiset₁ α) : multiset₁ α :=
quotient.lift_on₂ A B _ _
```

3c. Multisets over a type α are isomorphic to the function space $\alpha \to \mathbb{N}$, in the same way that sets over α are isomorphic to $\alpha \to$ `Prop`. Finite multisets correspond to functions with finite support. The *support* of a function `f` is defined formally as `{x | f x ≠ 0}`.

Use Lean's subtype mechanism to introduce an alternative definition of finite multisets by carving out the functions of finite support from the function space $\alpha \to \mathbb{N}$, by filling in the hole (_) below.

```
def multiset₂ (α : Type) : Type := _
```

Your definition may use the following predicate, which captures the notion of a finite set:

```
inductive finite {α : Type} : set α → Prop
| empty : finite ∅
| singleton (a : α) : finite (insert a ∅)
| union (A B : set α) (fin_A : finite A) (fin_B : finite B) : finite (A ∪ B)
```

3d. Implement the basic operations described in subquestion 3b on the subtype representation of multisets, by filling in the three holes (_) below. Supply any necessary proofs as well, informally, possibly by appealing to the inductive definition of `finite` provided above.

```
def empty_mset₂ {α : Type} : multiset₂ α := _
def singleton_mset₂ {α : Type} (a : α) : multiset₂ α := _
def sum_mset₂ {α : Type} (A B : multiset₂ α) : multiset₂ α := _
```

3e. Define a Lean function that converts the quotient representation of a multiset to its subtype representation, by filling in the two holes (_) below.

The second hole and parts of the first hole correspond to proof terms. For each of these, state the property that must be proved and explain very briefly why it should hold. You may assume reasonable lemmas about sets and the `finite` predicate.

```
def mset₂_of_mset₁ {α : Type} (A : multiset₁ α) : multiset₂ α :=
quotient.lift_on A _ _
```

Question 4. Hoare logic for DOWHILE language (5+5+5 points)

We start by defining the abstract syntax of two variants of the WHILE language. The first variant is the standard WHILE language as seen in class (but now called `wl`):

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

The second variant is a "DOWHILE" language:

```
inductive dl (σ : Type) : Type
| skip  {} : dl
| assign   : (σ → σ) → dl
| seq      : dl → dl → dl
| unless   : dl → (σ → Prop) → dl
| do_while : dl → (σ → Prop) → dl
```

The last two statements are new:

- `unless p c` executes `p` if `c` is *false* in the current state; otherwise it does nothing. This statement is inspired by Perl's `unless` conditional.
- `do_while p c` first executes `p`. Then, if `c` is true in the resulting state, it reenters the loop and executes `p` again, and continues executing `p` until `c` becomes false. The semantics is almost the same as `while c p`, except that `p` is always executed at least once, even if the condition is not true initially. This statement is inspired by C's and Java's `do { ... } while (...)` loop.

4a. Define a Lean function that translates a DOWHILE program to a WHILE program with the same big-step semantics, by completing the following definition:

```
def wl_of_dl (σ : Type) : dl σ → wl σ
```

4b. Give a sound and as-complete-as-possible partial correctness Hoare rule, or set of Hoare rules, for the DOWHILE language's `unless` statement. Briefly justify how you derived the rule(s). You may use the traditional rule format (with a horizontal bar) or Lean syntax to specify the rule(s).

Hint: One possible way to derive and justify your Hoare rule(s) is to apply the translation function developed in subquestion 4a and then to apply standard Hoare rules on the translation.

4c. Give a sound and as-complete-as-possible partial correctness Hoare rule, or set of Hoare rules, for the DOWHILE language's `do_while` statement. Briefly justify how you derived the rule(s). You may use the traditional rule format or Lean syntax to specify the rule(s).

Question 5. Denational semantics of LOOP and REPEAT languages (5+6+3 points)

5a. The "LOOP" language is identical to WHILE, except that its looping statement, called `loop`, executes its body an unspecified number of times (zero or more times).

```
inductive ll (σ : Type) : Type
| skip {} : ll
| assign  : (σ → σ) → ll
| seq     : ll → ll → ll
| ite     : (σ → Prop) → ll → ll → ll
| loop    : ll → ll
```

Complete the following relational denotational semantics, by filling in the two holes (_). You may use the transitive closure operator developed in Question 1 to specify the semantics of a loop (instead of a fixpoint operator).

```
def den (σ : Type) : ll σ → set (σ × σ)
| ll.skip        := _
| (ll.assign f)  := {st | st.2 = f st.1}
| (ll.seq p q)   := den p ○ den q
| (ll.ite c p q) := (den p ⇂ c) ∪ (den q ⇂ (λs, ¬ c s))
| (ll.loop p)    := _
```

5b. The "REPEAT" language is identical to WHILE, except that it executes its body a fixed number of times; for example, `repeat 3 p` executes `p` three times in sequence.

```
inductive rl (σ : Type) : Type
| skip {} : rl
| assign  : (σ → σ) → rl
| seq     : rl → rl → rl
| ite     : (σ → bool) → rl → rl → rl
| repeat  : ℕ → rl → rl
```

Complete the following functional denotational semantics, which executes a REPEAT program from an initial state, returning a final state. Do not worry yet about termination.

```
def run (σ : Type) : rl σ → σ → σ
| rl.skip       s := s
| (rl.assign f) s := f s
```

5c. Now you may worry about termination. Does the `run` function defined in subquestion 5b terminate? If yes, explain briefly why (e.g., by providing a well-founded relation or a measure that decreases with every recursive call). If no, provide a counterexample (e.g., a program, an initial state, and a brief explanation of what goes wrong).

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