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

Mock Exam
Thursday 32 Movember 2018, 15:15–18:00, WN-Y666
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.

Question 1. List concatenation and reversal (6+5+4+6 points)

Consider the following Lean function on lists:

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

1a. Prove the following lemma about concatenation:

```
lemma map_concat {α β : Type} (f : α → β) :
  ∀xss : list (list α), map f (concat xss) = concat (map (map f) xss)
```

You may assume basic lemmas about `map`, including its definition:

```
def map {α β : Type} (f : α → β) : list α → list β
| []        := []
| (x :: xs) := f x :: map xs
```

1b. Give the definition in Lean of a `reverse` function that takes a list in argument and returns it with its elements in reverse order.

1c. Complete the *statements* of the following properties that describes the behavior of `concat` and `reverse` (defined above) with respect to the append operator (`++`), by filling in the holes (`_`). These properties should describe how `concat` and `reverse` distribute over `++`. You are *not* asked to prove the properties.

```
lemma concat_snoc {α : Type} :
  ∀(xs : list α) (xss : list (list α)), concat (xss ++ [xs]) = _

lemma reverse_append {α : Type} :
  ∀xs ys : list α, reverse (xs ++ ys) = _
```

1d. Prove the following lemma, using the above definitions and properties.

```
lemma reverse_concat {α : Type} :
  ∀xss : list (list α), reverse (concat xss) =
    concat (reverse (map reverse xss))
```

Question 2. Sorted lists (6+4+4 points)

2a. Define in Lean an inductive predicate `sorted` that takes a list of natural numbers as argument and that returns `true` if and only if that list is sorted in increasing order. The definition should distinguish three cases:

- The empty list is sorted.
- Any list of length 1 is sorted.
- A list $[x_1, x_2, \ldots, x_n]$ of length $n \geq 2$ is sorted if $x_1 \leq x_2$ and $[x_2, \ldots, x_n]$ is sorted.

2b. Prove the following four lemmas to test your definition.

```
example : sorted []
example : sorted [2]
example : sorted [3, 5]
example : sorted [7, 9, 9, 11]
```

2c. Not all lists are sorted. Prove the following counterexample.

```
example : ¬ sorted [17, 13]
```

Question 3. Program equivalence (6+5+5 points)

Consider the following WHILE language:

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

In the following, we fix a type σ of states.

Recall the big-step semantic predicate `(p, s) ⟹ t` and the following notion of program equivalence:

```
def program_equiv (p₁ p₂ : program σ) : Prop :=
∀s t, (p₁, s) ⟹ t ↔ (p₂, s) ⟹ t
```

We use the abbreviation `p₁ ≈ p₂` for `program_equiv p₁ p₂`.

3a. Prove the following program equivalence, which can be used by a compiler to exchange the 'then' and the 'else' branches of an 'if' statement.

```
example {p q : program σ} {c : σ → Prop} :
  ite c p q ≈ ite (λs, ¬ c s) q p
```

3b. By appealing to the big-step semantics, explain why infinite loops never exit, a property that can be stated formally as follows.

```
lemma big_step_while_true {p : program σ} {s t : σ} :
  ¬ ⟨while (λ_, true) p, s⟩ ⟹ t
```

3c. Use the above lemma to prove the following program equivalence, which can be used to eliminate dead code in an optimizer.

```
example {p p' : program σ} : seq (while (λ_, true) p) p' ≈ while (λ_, true) p
```

Question 4. Finite sets (4+6+6 points)

4a. Define an inductive predicate `finite` in Lean that returns `true` if and only if its argument is a finite set (of type `set α`). The definition should distinguish two cases:

- The empty set is finite.
- If A is finite, then $\{a\} \cup A$ is finite.

Put the definition in a namespace called `v1`.

4b. Give an alternative Lean definition for `finite`, this time distinguishing three cases:

- The empty set is finite.
- The singleton set $\{a\}$ is finite.
- If A and B are finite, then $A \cup B$ is finite.

Put the definition in a namespace called `v2`.

4c. As a step towards proving the two definitions equivalent, prove the following lemma about `v1.finite`:

```
namespace v1

lemma finite.union {α : Type} :
  ∀A B : set α, finite A → finite B → finite (A ∪ B) :=

end v1
```

Question 5. Finite sets as quotients (6+3+5+5+4 points)

Finite sets can be defined in various ways. One way is to start with a raw type of finite lists and take the quotient over the equivalence relation `r := λxs ys, ∀x, x ∈ xs ↔ x ∈ ys`, meaning that two lists are put in the same equivalence class if they contain exactly the same elements.

In Lean, this could be done as follows:

```
instance fin_set.rel (α : Type) : setoid (list α) :=
{ r     := λxs ys, ∀x, x ∈ xs ↔ x ∈ ys,
  iseqv := _ }

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

However, there is a hole (`_`) in the definition, standing for a proof that the relation `r` is an equivalence relation (i.e., is reflexive, symmetric, and transitive).

5a. State and prove the three properties that capture the fact that `r` is an equivalence relation.

5b. The empty finite set can be defined in Lean as follows:

```
def fin_empty {α : Type} : fin_set α := ⟦[]⟧
```

Inspired by the above, define a `fin_singleton` operation that, given an element a, returns a singleton of type `fin_set α` containing that element.

5c. Next, define the union operator on `fin_set α` by filling the hole (`_`) in the following definition:

```
def fin_union {α : Type} (A B : fin_set α) : fin_set α :=
quotient.lift_on₂ A B (λas bs, ⟦as ++ bs⟧) _
```

The hole has type

```
∀as₁ as₂ bs₁ bs₂ : list α, r as₁ bs₁ → r as₂ bs₂ → ⟦as₁ ++ as₂⟧ = ⟦bs₁ ++ bs₂⟧
```

5d. Is the following property provable or not? If yes, please argue why. If no, give a counterexample (specifying concrete values for α and `A`).

```
lemma exists_list_of_fin_set {α : Type} (A : fin_set α) :
  ∃xs : list α, A = ⟦xs⟧
```

5e. Explain what the following definition achieves. Why do you think it might be useful?

```
noncomputable def list_of_fin_set {α : Type} (A : fin_set α) : list α :=
@classical.some (list α) (λxs, A = ⟦xs⟧) (exists_list_of_fin_set A)
```

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