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.

Answer:

This version of the exam includes suggested answers, presented in blocks like this one. Many of the proposed proofs are atypical in that they were developed using Lean and are syntactically correct. Such proofs would be accepted at the exam, but we realize that it is hard to develop correct Lean proofs on paper; hence the flexible proof guidelines above.

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
```

Answer:

```
| []          := by refl
| (xs :: xss) :=
  calc map f (concat (xs :: xss)) = map f (xs ++ concat xss) : by rw concat
  ... = map f xs ++ map f (concat xss) : by simp
  ... = map f xs ++ concat (map (map f) xss) : by rw map_concat
  ... = concat (map f xs :: map (map f) xss) : by rw concat
  ... = concat (map (map f) (xs :: xss)) : by simp
```

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.

Answer:

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

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) = _
```

Answer:

```
concat xss ++ xs
reverse ys ++ reverse xs
```

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))
```

Answer:

```
| []          := by refl
| (xs :: xss) :=
  calc reverse (concat (xs :: xss)) = reverse (xs ++ concat xss) : by refl
  ... = reverse (concat xss) ++ reverse xs : by rw reverse_append
  ... = concat (reverse (map reverse xss)) ++ reverse xs : by rw reverse_concat
  ... = concat (reverse (map reverse xss) ++ [reverse xs]) : by rw concat_snoc
  ... = concat (reverse (reverse xs :: map reverse xss)) : by rw reverse
  ... = concat (reverse (map reverse (xs :: xss))) : by refl
```

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.

Answer:

```
inductive sorted : list ℕ → Prop
| nil : sorted []
| single {x : ℕ} : sorted [x]
| two_or_more {x y : ℕ} {xs : list ℕ} (xy : x ≤ y) (yxs : sorted (y :: xs)) :
  sorted (x :: y :: xs)
```

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]
```

Answer:

```
example : sorted [] :=
  sorted.nil

example : sorted [2] :=
  sorted.single

example : sorted [3, 5] :=
  sorted.two_or_more dec_trivial sorted.single

example : sorted [7, 9, 9, 11] :=
  sorted.two_or_more dec_trivial
    (sorted.two_or_more dec_trivial
      (sorted.two_or_more dec_trivial
        sorted.single))
```

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

```
example : ¬ sorted [17, 13]
```

Answer:

```
assume s : sorted [17, 13],
have 17 ≤ 13 := match s with sorted.two_or_more xy yxs := xy end,
have ¬ (17 ≤ 13) := dec_trivial,
show false, from by contradiction
```

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
```

Answer:

```
begin
  intros s t,
  apply iff.intro,
  repeat {
    intro h,
    cases h,
    { apply big_step.ite_false; cc },
    { apply big_step.ite_true; cc } }
end
```

Intuitively, proving the equivalence amounts to proving two implications:

```
(ite c p q, s) ⟹ t → (ite (λs, ¬ c s) q p, s) ⟹ t
```

and

```
(ite (λs, ¬ c s) q p, s) ⟹ t → (ite c p q, s) ⟹ t
```

where `s` and `t` are fixed but unknown.

Let us focus on the first goal. The condition `c` is either true or false in state `s`. If it is true, then the hypothesis `(ite (λs, ¬ c s) q p, s) ⟹ t` must have been proved using the `ite_true` rule from the big-step semantics, and we apply the `ite_false` rule to prove the desired conclusion. This works because `¬¬ c s`

is true if `c s` is true. If the condition is false, the hypothesis must have been proved using `ite_false`, and we apply `ite_true` to prove the conclusion, analogously.

The case of the second goal is completely analogous and therefore omitted.

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
```

Answer:

```
begin
  generalize eq : (while (λ_, true) p, s) = qs,
  intro h,
  induction h generalizing s; cases eq,
  { apply h_ih_hw, refl },
  { apply h_hs, trivial }
end
```

Admittedly, the above is not very readable. The idea is to assume `⟨while (λ_, true) p, s⟩ ⟹ t` and show that this leads to a contradiction. We perform an induction on the rules of the big-step semantics. The only two possible cases, syntactically, are the case where the condition `(λ_, true)` is true in `s` and the case where it is false. Clearly, the second case is impossible (`(λ_, true) s = true`). Remains the first case.

By the big-step semantics, there must exist a state `u` such that

$$\langle \texttt{p, s} \rangle \Longrightarrow \texttt{u} \quad \text{and} \quad \langle \texttt{while } (\lambda\_, \texttt{ true) p, u} \rangle \Longrightarrow \texttt{t}.$$

But by the (second) induction hypothesis, we also know that

$$\neg \langle \texttt{while } (\lambda\_, \texttt{ true) p, u} \rangle \Longrightarrow \texttt{t}.$$

Contradiction.

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
```

Answer:

```
begin
  intros s t,
  apply iff.intro,
  { intro h, cases h, exact absurd h_h₁ big_step_while_true },
  { intro h, exact absurd h big_step_while_true }
end
```

Intuitively, to prove the property, we must prove two implications:

$$\texttt{(seq (while (}\lambda\texttt{\_, true) p) p', s)} \Longrightarrow \texttt{t} \rightarrow \texttt{(while (}\lambda\texttt{\_, true) p, s)} \Longrightarrow \texttt{t}$$

and

$$\texttt{(while (}\lambda\texttt{\_, true) p, s)} \Longrightarrow \texttt{t} \rightarrow \texttt{(seq (while (}\lambda\texttt{\_, true) p) p', s)} \Longrightarrow \texttt{t}$$

The second goal is easier. From the assumption $\texttt{while (}\lambda\texttt{\_, true) p, s)} \Longrightarrow \texttt{t}$, we immediately have a contradiction with the lemma `big_step_while_true` proved above.

For the first goal, from the assumption $\texttt{seq (while (}\lambda\texttt{\_, true) p) p', s)} \Longrightarrow \texttt{t}$, we obtain the existence of a state `u` such that

$$\texttt{(while (}\lambda\texttt{\_, true) p, s)} \Longrightarrow \texttt{u} \quad \text{and} \quad \texttt{(p', u)} \Longrightarrow \texttt{t}$$

The first property contradicts `big_step_while_true`.

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`.

Answer:

```
namespace v1

inductive finite {α : Type} : set α → Prop
| empty : finite ∅
| insert (a : α) (A : set α) (fin_A : finite A) : finite (insert a A)

end v1
```

(Instead of `insert a A`, we would also accept the traditional syntax `{a} ∪ A`.)

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`.

Answer:

```
namespace v2

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)

end 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
```

Answer:

```
namespace v1

-- basic lemmas we assume
@[simp] lemma union_empty_left {α : Type} {A : set α} :
  ∅ ∪ A = A := sorry
@[simp] lemma insert_union {α : Type} {a : α} {A B : set α} :
  insert a A ∪ B = insert a (A ∪ B) := sorry

lemma finite.union {α : Type} :
  ∀A B : set α, finite A → finite B → finite (A ∪ B) :=
begin
  intros A B fin_A fin_B,
  induction fin_A,
  { simp,
    exact fin_B },
  { simp,
    apply finite.insert,
    exact fin_A_ih }
end

end v1
```

Intuitively, we perform an induction on `finite A`. There are two cases.

If `A` is empty, then we have `A ∪ B = B`; hence the goal is simply `finite B`, which we have by assumption (`fin_B`).

If `A` is of the form `{a} ∪ A'`, then the goal is to prove `finite ({a} ∪ A') ∪ B`. The induction hypothesis tells us that `finite (A' ∪ B)`.

Clearly, we have `({a} ∪ A') ∪ B = {a} ∪ (A' ∪ B)`. We can use the second introduction rule for `finite` (called `finite.insert` above) to prove this, leaving the proof obligation `finite (A' ∪ B)`. This is exactly the induction hypothesis.

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.

Answer:

The properties can be stated as follows:

```
lemma r_reflexive (xs : list α) : r xs xs
lemma r_symmetric (xs ys : list α) : r xs ys → r ys xs
lemma r_transitive (xs ys zs : list α) : r xs ys → r ys zs → r xs zs
```

By expanding the definition of `r`, we basically exploit the fact that ↔ is itself an equivalence relation, proving each of the three properties of `r` by appealing to the analogous property of ↔.

For those who care, the hole in the `fin_set.rel` proof above can be filled as follows:

```
⟨ (assume xs, by simp),
  (assume xs ys ys_xs x, by simp *),
  (assume xs ys zs xs_ys ys_zs, by simp *) ⟩
```

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.

Answer:

```
def fin_singleton {α : Type} (a : α) : fin_set α := ⟦[a]⟧
```

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

$$\forall \mathtt{as}_1\ \mathtt{as}_2\ \mathtt{bs}_1\ \mathtt{bs}_2 : \mathtt{list}\ \alpha,\ \mathtt{r}\ \mathtt{as}_1\ \mathtt{bs}_1 \to \mathtt{r}\ \mathtt{as}_2\ \mathtt{bs}_2 \to [\![\mathtt{as}_1\ \mathtt{++}\ \mathtt{as}_2]\!] = [\![\mathtt{bs}_1\ \mathtt{++}\ \mathtt{bs}_2]\!]$$

Answer:

By soundness of quotients (`quotient.sound`), it suffices to prove that

$$\mathtt{r}\ (\mathtt{as}_1\ \mathtt{++}\ \mathtt{as}_2)\ (\mathtt{bs}_1\ \mathtt{++}\ \mathtt{bs}_2)$$

By unfolding the definition of `r`, we obtain the goal

$$\mathtt{x} \in \mathtt{as}_1\ \mathtt{++}\ \mathtt{as}_2 \leftrightarrow \mathtt{x} \in \mathtt{bs}_1\ \mathtt{++}\ \mathtt{bs}_2$$

In the left-to-right direction, we have $\mathtt{x} \in \mathtt{as}_1\ \mathtt{++}\ \mathtt{as}_2$ and must prove $\mathtt{x} \in \mathtt{bs}_1\ \mathtt{++}\ \mathtt{bs}_2$. By case analysis on the hypothesis, we have either $\mathtt{x} \in \mathtt{as}_1$ or $\mathtt{x} \in \mathtt{as}_2$. In the first subcase, we prove $\mathtt{x} \in \mathtt{bs}_1$ by exploiting the hypothesis $\mathtt{r}\ \mathtt{as}_1\ \mathtt{bs}_1$; hence $\mathtt{x} \in \mathtt{bs}_1\ \mathtt{++}\ \mathtt{bs}_2$. The second subcase is analogous but first proves $\mathtt{x} \in \mathtt{bs}_2$.

The right-to-left direction is entirely analogous.

In Lean:

```
begin
  intros as₁ bs₁ as₂ bs₂ as₁_bs₁ as₂_bs₂,
  apply quotient.sound,
  intro x,
  simp,
  apply iff.intro,
  { intro h,
    cases h,
    { apply or.intro_left,
      rw ←as₁_bs₁ x,
      assumption },
    { apply or.intro_right,
      rw ←as₂_bs₂ x,
      assumption } },
  { intro h,
    cases h,
    { apply or.intro_left,
      rw as₁_bs₁ x,
      assumption },
    { apply or.intro_right,
      rw as₂_bs₂ x,
      assumption } }
end
```

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⟧
```

Answer:

The property is provable. Without loss of generality, `A` can be expressed as ⟦`ys`⟧ for some list `ys`. (In Lean, this step appeals to the misnamed lemma `quotient.induction_on`.) We can then take `ys` as the witness for `xs`. The proof is then by reflexivity of `=`.

In Lean:

```
lemma exists_list_of_fin_set {α : Type} (A : fin_set α) :
  ∃xs : list α, A = ⟦xs⟧ :=
begin
  apply quotient.induction_on A,
  intro,
  apply exists.intro,
  refl
end
```

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)
```

Answer:

The function converts a finite set `A` into an arbitrary list that contains the same element. It can be useful to repair mismatches in APIs (for example, if an API requires a list but all we have is a finite set).

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