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 predi-
cate, 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:

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

1a. Prove the following lemma about concatenation:

```lean
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:

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

**Answer:**

```lean
| [] := 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:**

```lean
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.

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

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

click 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:

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

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

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

Answer:

```lean
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:

\[
\text{inductive program } (\sigma : \text{Type}) : \text{Type} \\
| \text{skip } \{\} : \text{program} \\
| \text{assign} : (\sigma \to \sigma) \to \text{program} \\
| \text{seq} : \text{program} \to \text{program} \to \text{program} \\
| \text{ite} : (\sigma \to \text{Prop}) \to \text{program} \to \text{program} \to \text{program} \\
| \text{while} : (\sigma \to \text{Prop}) \to \text{program} \to \text{program} \to \text{program}
\]

In the following, we fix a type \(\sigma\) of states.

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

\[
\text{def program_equiv } (p_1 : \text{program } \sigma, p_2 : \text{program } \sigma) : \text{Prop} := \\
\forall s t, (p_1, s) \Rightarrow t \iff (p_2, s) \Rightarrow t
\]

We use the abbreviation \(p_1 \approx p_2\) for \(\text{program_equiv } p_1 p_2\).

**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.

\[
\text{example } \{p \text{ q : program } \sigma\} \{c : \sigma \to \text{Prop}\} : \\
\text{ite } c \text{ p q } \approx \text{ite } (\lambda s, \neg c s) \text{ q p}
\]

**Answer:**

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

Intuitively, proving the equivalence amounts to proving two implications:

\[
(\text{ite } c \text{ p q, s} \Rightarrow t \to (\text{ite } (\lambda s, \neg c s) \text{ q p, s}) \Rightarrow t)
\]

and

\[
(\text{ite } (\lambda s, \neg c s) \text{ q p, s} \Rightarrow t \to \text{ite } c \text{ p q, s}) \Rightarrow t
\]

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 \(\text{ite } (\lambda s, \neg c s) \text{ q p, s} \Rightarrow t\) must have been proved using the \text{ite_true} rule from the big-step semantics, and we apply the \text{ite_false} rule to prove the desired conclusion. This works because \(\neg c s\)
is true if \( c \) \( s \) is true. If the condition is false, the hypothesis must have been proved using \texttt{ite_false}, and we apply \texttt{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.

\[
\text{lemma big_step_while_true \{p : program \sigma\} \{s t : \sigma\} :} \\
\neg \langle \text{while } (\lambda_-, \text{true}) p, s \rangle = \Rightarrow t
\]

\textbf{Answer:}

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

Admittedly, the above is not very readable. The idea is to assume \langle \text{while } (\lambda_-, \text{true}) p, s \rangle = \Rightarrow 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 \( (\lambda_-, \text{true}) \text{ is true in } s \) and the case where it is false. Clearly, the second case is impossible \((\lambda_-, \text{true}) \text{ s } = \text{true})\). Remains the first case.

By the big-step semantics, there must exist a state \( u \) such that

\[ \langle p, s \rangle = \Rightarrow u \quad \text{and} \quad \langle \text{while } (\lambda_-, \text{true}) p, u \rangle = \Rightarrow t. \]

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

\[ \neg \langle \text{while } (\lambda_-, \text{true}) p, u \rangle = \Rightarrow t. \]

\textbf{Contradiction.}

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

\texttt{example \{p p' : program \sigma\} : seq (while (\lambda_-, \text{true}) p) p' \approx while (\lambda_-, \text{true}) p}

\textbf{Answer:}

\begin{verbatim}
begin
  intros s t,
  apply iff.intro,
  { intro h, cases h, exact absurd h_h, big_step_while_true },
  { intro h, exact absurd h big_step_while_true }
end
\end{verbatim}
Intuitively, to prove the property, we must prove two implications:

\[(\text{seq} \ (\text{while} \ (\lambda_1, \text{true}) \ p) \ p', \ s) \implies t \to (\text{while} \ (\lambda_1, \text{true}) \ p, \ s) \implies t\]

and

\[(\text{while} \ (\lambda_1, \text{true}) \ p, \ s) \implies t \to (\text{seq} \ (\text{while} \ (\lambda_1, \text{true}) \ p) \ p', \ s) \implies t\]

The second goal is easier. From the assumption \((\text{while} \ (\lambda_1, \text{true}) \ p, \ s) \implies t\), we immediately have a contradiction with the lemma \text{big_step_while_true} proved above.

For the first goal, from the assumption \((\text{seq} \ (\text{while} \ (\lambda_1, \text{true}) \ p) \ p', \ s) \implies t\), we obtain the existence of a state \(u\) such that

\[(\text{while} \ (\lambda_1, \text{true}) \ p, \ s) \implies u \quad \text{and} \quad (p', \ u) \implies t\]

The first property contradicts \text{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:**

```lean
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 \( \{a\} \cup A \).)

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:**

```lean
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`:

```lean
namespace v1

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

9
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 \( r := \lambda x s \ y s, \ \forall x, \ x \in x s \leftrightarrow x \in y s \), 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:

```lean
instance fin_set.rel (α : Type) : setoid (list α) :=
{ r := \lambda x s y s, \ \forall x, \ x \in x s \leftrightarrow x \in y s,
  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:

```lean
lemma r_reflexive (xs : list α) : r xs xs
lemma r_symmetric (xs ys : list α) : r xs ys \rightarrow r ys xs
lemma r_transitive (xs ys zs : list α) : r xs ys \rightarrow r ys zs \rightarrow r xs zs
```

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

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

```lean
⟨ (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:

```lean
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 \( \text{fin_set } \alpha \) containing that element.

**Answer:**

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

5c. Next, define the union operator on \( \text{fin_set } \alpha \) by filling the hole (\_) in the following definition:

```lean
def fin_union {α : Type} (A B : fin_set α) : fin_set α :=
quotient.lift_on2 A B (\lambda bs, [[bs]]) _
```
The hole has type

\[ \forall \alpha. r \ as_1 as_2 \ bs_1 \ bs_2 : list \alpha, r \ as_1 \ bs_1 \rightarrow r \ as_2 \ bs_2 \rightarrow [as_1 \ ++ \ as_2] = [bs_1 \ ++ \ bs_2] \]

**Answer:**

By soundness of quotients (\texttt{quotient.sound}), it suffices to prove that

\[ r \ (as_1 \ ++ \ as_2) \ (bs_1 \ ++ \ bs_2) \]

By unfolding the definition of \( r \), we obtain the goal

\[ x \in as_1 \ ++ \ as_2 \leftrightarrow x \in bs_1 \ ++ \ bs_2 \]

In the left-to-right direction, we have \( x \in as_1 \ ++ \ as_2 \) and must prove \( x \in bs_1 \ ++ \ bs_2 \). By case analysis on the hypothesis, we have either \( x \in as_1 \) or \( x \in as_2 \). In the first subcase, we prove \( x \in bs_1 \) by exploiting the hypothesis \( r \ as_1 \ bs_1 \); hence \( x \in bs_1 \ ++ \ bs_2 \). The second subcase is analogous but first proves \( x \in bs_2 \).

The right-to-left direction is entirely analogous.

In Lean:

```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 \( \alpha \) 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:

```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?

```lean
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).