Institute for Informatics of the Ludwig-Maximilians-Universität München
Yiming Xu, Ph.D.
Prof. Dr. Jasmin Blanchette
Alexandra Graß

Summer semester 2026
21.07.2026

Solution to the Regular Examination in the Course
Interactive Theorem Proving

You have **120 minutes** at your disposal. Written or electronic aids are not permitted except for normal watches. Carrying forbidden devices, even switched off, will be considered a cheating attempt.

Write your full name and matriculation number clearly legible on this cover sheet, as well as your name in the header on each sheet. Hand in all sheets. Leave them stapled together. Use only **pens** and **neither** the color **red nor green**.

Check that you have received all the sheets. Guidelines for writing pen-and-paper proofs are given on **page 2**. Questions can be found on **pages 3–17**. There are 7 questions for a total of 100 points. You may use the back of the sheets for secondary calculations. If you use the back of a sheet to answer, clearly mark what belongs to which question and indicate in the corresponding question where all parts of your answer can be found. Cross out everything that should not be graded.

With your signature, you confirm that you are sufficiently healthy at the beginning of the examination and that you accept the examination bindingly.

Last name (in CAPITAL LETTERS):

First name (in CAPITAL LETTERS):

Matriculation number:

Program of study:

Hierby I confirm the correctness of the above information: _____
Signature

Please leave the following table blank:

Question	1	2	3	4	5	6	7	$\sum$
Points	23	14	16	13	16	7	11	100
Score								

Guidelines for Paper Proofs

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, including 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 induction hypotheses assumed (if any) and the goal to be proved. Minor proof steps corresponding to `refl`, `simp`, or `linarith` need not be justified if you think they are obvious, but you should mention which key theorems they depend on. You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate.

Solution to Question 1 (Types and Terms): **(23 points)**

a) [14 points] Recall the following simplified typing rules for Lean's dependent type theory:

$$\frac{}{\texttt{C} \vdash \texttt{c} : \sigma}\ \textsc{Cst} \quad \text{if } \texttt{c} \text{ is globally declared with type } \sigma$$

$$\frac{}{\texttt{C} \vdash \texttt{x} : \sigma}\ \textsc{Var} \quad \text{if } \texttt{x} : \sigma \text{ is the rightmost occurrence of } \texttt{x} \text{ in } \texttt{C}$$

$$\frac{\texttt{C} \vdash \texttt{t} : (\texttt{x} : \sigma) \to \tau\texttt{[x]} \quad \texttt{C} \vdash \texttt{u} : \sigma}{\texttt{C} \vdash \texttt{t u} : \tau\texttt{[u]}}\ \textsc{App}'$$

$$\frac{\texttt{C, x} : \sigma \vdash \texttt{t} : \tau\texttt{[x]}}{\texttt{C} \vdash \texttt{(fun x} : \sigma \mapsto \texttt{t)} : (\texttt{x} : \sigma) \to \tau\texttt{[x]}}\ \textsc{Fun}'$$

Let $\texttt{a} : \mathbb{N}$, $\texttt{b} : \mathbb{N}$, $\texttt{c} : \mathbb{N}$, $\texttt{P} : \mathbb{N} \to \mathbb{N} \to \texttt{Prop}$, and $\texttt{Nat.add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ be globally declared constants. What is the type of the following two Lean terms? Give in each case a typing derivation as justification for the type.

(i) `P (Nat.add a b) c` (8 points)

PROPOSED SOLUTION: The type is `Prop`. The typing derivation is

$$\dfrac{\dfrac{\texttt{P}: \mathbb{N} \to \mathbb{N} \to \texttt{Prop}}{\vdash \texttt{P} : \mathbb{N} \to \mathbb{N} \to \texttt{Prop}}\textsc{Cst} \quad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{Nat.add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}}\textsc{Cst} \quad \dfrac{}{\vdash \texttt{a} : \mathbb{N}}\textsc{Cst}}{\vdash \texttt{Nat.add a} : \mathbb{N} \to \mathbb{N}}\textsc{App}' \quad \dfrac{}{\vdash \texttt{b} : \mathbb{N}}\textsc{Cst}}{\vdash \texttt{Nat.add a b} : \mathbb{N}}\textsc{App}'}{\vdash \texttt{P (Nat.add a b)} : \mathbb{N} \to \texttt{Prop}}\textsc{App}' \quad \dfrac{}{\vdash \texttt{c} : \mathbb{N}}\textsc{Cst}}{\vdash \texttt{P (Nat.add a b) c} : \texttt{Prop}}\textsc{App}'$$

- 2 points for type
- 6 points for derivation tree

(ii) `fun (n : ℕ) ↦ fun (m : ℕ) ↦ n` (6 points)

PROPOSED SOLUTION: The type is $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. The typing derivation is

$$\dfrac{\dfrac{\dfrac{}{\texttt{n} : \mathbb{N}, \texttt{m} : \mathbb{N} \vdash \texttt{n} : \mathbb{N}}\textsc{Var}}{\texttt{n} : \mathbb{N} \vdash \texttt{(fun m} : \mathbb{N} \mapsto \texttt{n)} : \mathbb{N} \to \mathbb{N}}\textsc{Fun}'}{\vdash \texttt{(fun n} : \mathbb{N} \mapsto \texttt{fun m} : \mathbb{N} \mapsto \texttt{n)} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}}\textsc{Fun}'$$

- 2 points for type
- 4 points for derivation tree

b) [9 points] Let α, β, and γ be Lean types. Give an inhabitant for each of the following types:

(i) `(`$\alpha \to \beta \to \gamma$`)` $\to \beta \to \alpha \to \gamma$ (3 points)

PROPOSED SOLUTION: `fun (f:` $\alpha \to \beta \to \gamma$`) (b :` β`) (a :` α`)` $\mapsto$ `f a b`

- 1 point for `fun f b a`
- 2 points for the output term

(ii) `(`$\alpha \to \beta$`)` $\to \alpha \to$ `(`$\beta \to \gamma$`)` $\to$ `(`$\alpha \to \gamma$`)` (3 points)

PROPOSED SOLUTION: `fun (f :` $\alpha \to \beta$`) (a :` α`) (g :` $\beta \to \gamma$`) (a' :` α`)` $\mapsto$ `g (f a)`

- 1 point for `fun f a g a'`
- 2 points for the output term

(iii) $\alpha \to \mathbb{N}$ (3 points)

PROPOSED SOLUTION: `fun a :` $\alpha \mapsto$ `0`

- 1 point for `fun a :` α
- 2 points for the output number

Solution to Question 2 (Functional Programming): **(14 points)**

a) [8 points] Consider the following Lean function definition:

```
def replicate {α : Type} (a : α) : ℕ → List α
  | 0 => []
  | Nat.succ n => a :: replicate a n
```

Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 2. (8 points)

```
theorem replicate_length {α : Type} (a : α) (n : ℕ) :
    List.length (replicate a n) = n
```

PROPOSED SOLUTION: The proof is by structural induction on `n`.

The base case is

$$\texttt{List.length (replicate a 0)} = \texttt{0}$$

Both sides simplify to `0` and are hence equal.

The induction step is

$$\texttt{List.length (replicate a (n + 1))} = \texttt{n} + 1$$

The induction hypothesis is

$$\texttt{List.length (replicate a n)} = \texttt{n}$$

The induction step simplifies to

$$\texttt{List.length (replicate a n)} + 1 = \texttt{n} + 1$$

By the induction hypothesis, the two sides are equal.

- 1 point for "by induction"
- 1 point for "on `n`"
- 1 point for statement of base case
- 1 point for proof of base case
- 1 point for statement of induction step
- 1 point for statement of induction hypothesis
- 2 points for proof of induction step (`simp` and IH)

b) [6 points] Define a polymorphic Lean function `constantSquareMatrix` that replicates a value in two dimensions. For example, `constantSquareMatrix 1 5` should evaluate to `[[5]]` and `constantSquareMatrix 3 "a"` should evaluate to `[["a", "a", "a"], ["a", "a", "a"], ["a", "a", "a"]]`. (6 points)

PROPOSED SOLUTION:

```
def constantSequareMatrix {α : Type} (n : ℕ) : α → List (List α) :=
  fun a ↦ replicate (replicate a n) n
```

- 2 points for "`constantSequareMatrix {α : Type} (n : ℕ)`"
- 2 points for "`α → List (List α)`"
- 2 points for "`fun a ↦ replicate (replicate a n) n`"

Solution to Question 3 (Inductive Predicates): (16 points)

a) [6 points] Consider the following Lean inductive predicate, which determines whether a list is decreasing with a difference of `2` between consecutive items:

```
inductive IsDiffTwoSeq: List ℕ → Prop
  | nil : IsDiffTwoSeq []
  | sing (n : ℕ) : IsDiffTwoSeq [n]
  | add_two (a : ℕ) (l : List ℕ) (h : IsDiffTwoSeq (a :: l)) :
      IsDiffTwoSeq ((a + 2) :: a :: l)
```

For example, `IsDiffTwoSeq [14, 12, 10]` should hold, whereas `IsDiffTwoSeq [3, 2, 1]` should not hold.

Prove the following Lean theorem about `IsDiffTwoSeq`. If you want to use the inductive rules for defining `IsDiffTwoSeq`, write down the concrete values `n`, `a`, and `l` to be instantiated to the variables. Make sure to follow the proof guidelines given on page 2. (6 points)

```
theorem IsDiffTwoSequence_example : IsDiffTwoSeq [122, 120]
```

PROPOSED SOLUTION:

The proof is by applying the inductive rules in the definition of `IsDiffTwoSeq`.

By applying `IsDiffTwoSeq.add_two` to the initial goal, with `a = 120` and `l = []`, the goal is reduced to:

```
IsDiffTwoSeq [120]
```

which holds by `IsDiffTwoSeq.sing`, with `n = 120`.

- 2 points for `IsDiffTwoSeq.add_two`
- 1 point for `a = 120`
- 1 point for `l = []`
- 1 point for `IsDiffTwoSeq.sing`
- 1 point for `n = 120`

b) [10 points] State the following propositions in Lean. Indicate whether they are theorems or not.

(i) If two lists both satisfy the predicate `IsDiffTwoSeq`, then concatenating them produces a list satisfying the predicate `IsDiffTwoSeq`.

(ii) Reversing a list satisfying the predicate `IsDiffTwoSeq` produces a list satisfying the predicate `IsDiffTwoSeq`. You can use the `List.reverse` function.

In each case, give a brief proof or counterexample.

(10 points)

PROPOSED SOLUTION:

```
∀xs ys, IsDiffTwoSeq xs ∧ IsDiffTwoSeq ys → IsDiffTwoSeq (xs ++ ys)
```

Counterexample: Any one-item-list satisfies `IsDiffTwoSeq`, but `[2] ++ [0]` does not.

```
∀xs, IsDiffTwoSeq xs → IsDiffTwoSeq (List.reverse xs)
```

Counterexample: The list `[2, 0]` satisfies `IsDiffTwoSeq`, but `[0, 2]` does not.

- 1 point for quantifiers or variables for `xs` and `ys`
- 1 point for `IsDiffTwoSeq xs ∧ IsDiffTwoSeq ys`
- 1 point for the implication
- 1 point for `IsDiffTwoSeq (xs ++ ys)`
- 1 point for a counterexample
- 1 point for quantifiers or variables for `xs`
- 1 point for `IsDiffTwoSeq xs`
- 1 point for the implication
- 1 point for `IsDiffTwoSeq (List.reverse xs)`
- 1 point for a counterexample

Solution to Question 4 (Effectful Programming): **(13 points)**

During the execution of a program, we may want to gather all the error messages together to print them all at the very end. The *console* monad records all the error messages along the whole execution. In Lean, the console monad can be defined as follows:

```
def Console (α : Type) : Type := α × String
def Console.value {α : Type} : Console α → α := Prod.fst
def Console.message {α : Type} : Console α → String := Prod.snd

def Console.pure {α : Type} : α → Console α := fun a ↦ (a, "")
def Console.bind {α β : Type} : Console α → (α → Console β) → Console β :=
  fun ma f ↦
    let mb := f (Console.value ma)
    (Console.value mb, Console.message ma ++ Console.message mb)

instance Console.Pure : Pure Console :=
  { pure := Console.pure }

instance Console.Bind : Bind Console :=
  { bind := Console.bind }
```

a) [6 points] Prove the following law about the console monad. Make sure to follow the proof guidelines given on page 2. In addition, show all the steps when unfolding the definition of monad operators. (6 points)

```
lemma Console.pure_bind {α β : Type} (a : α) (f : α → Console β) :
    Console.pure a >>= f = f a
```

PROPOSED SOLUTION: The following sequence of equalities proves the theorem:

```
calc (pure a >>= f)
    = match Console.pure a with
      | (a', str) => ((f a').value, str ++ (f a').message)
      := by rfl
    _ = match (a, "") with
      | (a' , str) => ((f a').value, str ++ (f a').message) := by rfl
    _ = ((f a).value, "" ++ (f a).message) := by rfl
    _ = ((f a).value, (f a).message) := by rfl
    _ = f a := by rfl
```

- 2 points for unfolding of `bind`
- 2 points for unfolding of `pure`
- 2 points for simplification

b) [3 points] We can define a function in Lean

```
def Console.log {α : Type} : Console α → String → Console α :=
  fun (a, m) m' ↦ (a, m ++ m')
```

that adds a message after a value in `Console` α. Evaluate the following expression:

```
Console.bind (Console.pure 3)
  (fun a ↦ Console.log (Console.pure a) "multiplication by 7 detected")
```

PROPOSED SOLUTION: The evaluation output is

```
(3, "multiplication by 7 detected")
```

- 1 point for `3`
- 1 point for `"multiplication by 7 detected"`
- 1 point for putting them into a pair.

c) [4 points] Commutative monads are monads for which we can reorder actions that do not depend on each other. For the console monad, the property can be stated as follows:

```
theorem Console.bind_comm {α β γ δ : Type} (ma : Console α)
  (f : α → Console β) (g : α → Console γ) (h : α → β → γ → Console δ) :
    ma >>= (fun a ↦ f a >>= fun b ↦ g a >>= fun c ↦ h a b c) =
    ma >>= (fun a ↦ g a >>= fun c ↦ f a >>= fun b ↦ h a b c)
```

Is `Console` a commutative monad? If yes, provide a justification (as text or in the form of a brief proof sketch). If no, provide a counterexample (i.e., concrete values for `ma`, `f`, `g`, and `h` above and the result of evaluating both sides of the equality).

PROPOSED SOLUTION: No, it is not a commutative monad. A counterexample is

```
def f : ℕ → Console ℕ := fun n ↦ (n, "m1")
def g : ℕ → Console ℕ := fun n ↦ (n, "m2")
def h : ℕ → ℕ → ℕ → Console ℕ := fun a ↦ fun b ↦ fun c ↦ pure a
```

- 1 point for "no".
- 3 points for the counterexample.

Solution to Question 5 (Operational Semantics): (16 points)

The INTER programming language is similar to the familiar WHILE language, with the `while`–`do` statement replaced by an `interleave` statement. The `interleave` statement takes a natural number `n` and two statements `S` and `T`, and performs `n` iterations, alternating `S` and `T`. For example, the `INTER` program

```
interleave 3 (j := j + 1) (i := i + 2)
```

adds `1` to `j`, adds `2` to `i`, and adds `1` to `j` again.

In Lean, the INTER syntax is modeled abstractly by the following datatype:

```
inductive Stmt : Type where
  | skip       : Stmt
  | assign     : String → (State → ℕ) → Stmt
  | seq        : Stmt → Stmt → Stmt
  | ifThen     : (State → Prop) → Stmt → Stmt
  | interleave : ℕ → Stmt → Stmt → Stmt

infixr:90 "; " => Stmt.seq
```

a) [8 points] Complete the following specification of a small-step semantics for INTER by giving the missing derivation rules for `interleave`. (8 points)

$$\frac{}{\texttt{(x := a, s)} \Rightarrow \texttt{(skip, s[x} \mapsto \texttt{s(a)])}}\ \textsc{Assign}$$

$$\frac{\texttt{(S, s)} \Rightarrow \texttt{(S', s')}}{\texttt{(S; T, s)} \Rightarrow \texttt{(S'; T, s')}}\ \textsc{Seq-Step} \qquad \frac{}{\texttt{(skip; T, s)} \Rightarrow \texttt{(T, s)}}\ \textsc{Seq-Skip}$$

$$\frac{}{\texttt{(ifThen B S, s)} \Rightarrow \texttt{(S, s)}}\ \textsc{If-True} \quad \text{if } \texttt{s(B)} \text{ is true}$$

$$\frac{}{\texttt{(ifThen B S, s)} \Rightarrow \texttt{(skip, s)}}\ \textsc{If-False} \quad \text{if } \texttt{s(B)} \text{ is false}$$

PROPOSED SOLUTION:

$$\frac{}{\texttt{(interleave 0 S T, s)} \Rightarrow \texttt{(skip, s)}}\ \textsc{interleave-zero}$$

$$\frac{}{\texttt{(interleave (n + 1) S T, s)} \Rightarrow \texttt{(S ; interleave n T S, s)}}\ \textsc{interleave-succ}$$

- 3 points for INTERLEAVE-ZERO
  - 2 points for the LHS
  - 1 point for the RHS
- 5 points for INTERLEAVE-SUCC
  - 2 points for the LHS
  - 3 points for the RHS

b) [8 points] Recall that for the inductive predicate

```
inductive SmallStep : Stmt × State → Stmt × State → Prop where
```

the rule for sequential composition (`;`) is encoded in Lean as

```
| seq_step (S S' T s s') (hS : SmallStep (S, s) (S', s')) :
      SmallStep (S; T, s) (S'; T, s')
| seq_skip (T s) :
      SmallStep (Stmt.skip; T, s) (T, s)
```

Encode the rules for `ifThen` in a Lean definition of an inductive predicate that encodes the small-step semantics based on subquestion a) above. You are not asked to provide any rules for any other constructor. (8 points)

PROPOSED SOLUTION:

```
| if_true (B S s) (hcond : B s) :
  SmallStep (Stmt.ifThen B S, s) (S, s)
| if_false (B S s) (hcond : ¬ B s) :
  SmallStep (Stmt.ifThen B S, s) (Stmt.skip, s)
```

- 4 points for `if_true`
  - 2 points for rule name, variables, and condition
  - 2 points for conclusion
- 4 points for `if_false`
  - 2 points for rule name, variables, and condition
  - 2 points for conclusion

Solution to Question 6 (Logic): **(7 points)**

One of the following statements is a theorem; the other one is not. Prove the theorem.

`(∀y, ∃x, P x y) → ∃x, ∀y, P x y`

`(∃x, ∀y, P x y) → ∀y, ∃x, P x y`

In your proof, identify clearly the quantifier reasoning. Make sure to follow the proof guidelines given on page 2.

PROPOSED SOLUTION: The theorem is

`(∃x, ∀y, P x y) → ∀y, ∃x, P x y`

Assume `∃x, ∀y, P x y`. This means that there exists a witness `w` such that for each `y`, we have `P w y`. To prove `∀y, ∃ x, P x y`, we fix `y` to be an arbitrary value. Now we need to prove `∃x, P x y`. We can use `w` as the witness for the quantified `x`. It remains to prove `P w y`, which we have as an assumption about `w`.

- 2 points for picking the correct theorem
- 5 points for the proof
  - 4 points for reasoning about each quantifier
  - 1 point for the flow of the argument.

Solution to Question 7 (Foundations): **(11 points)**

a) [4 points] Let σ `: Type 3` be a Lean type. Give the type of each of the following Lean terms.

$\mathbb{N} \to$ `Prop`
$\sigma \to \mathbb{N}$
`Sort 3`
`fun` (α `: Type`) $\mapsto$ `fun` (β `: Type`) $\mapsto$ β

PROPOSED SOLUTION:
`Type`
`Type 3` or `Sort 4`
`Type 3`
`Type` $\to$ `Type` $\to$ `Type`

- 1 point per type

b) [7 points] The set of natural numbers less than `n` can be defined in Lean as a subtype of ℕ:

```
def Finite (n : ℕ) : Type :=
  { m : ℕ // m < n}
```

We can define an operation on the subtype as follows:

```
def Finite.succ {n : ℕ} (a : Finite n) :
    Finite (n + 1) :=
  Subtype.mk (Subtype.val a + 1)
  (by
     apply Nat.succ_lt_succ
     simp[Subtype.property a])
```

Inspired by the definition of `Finite.succ`, define the addition on elements of the type `Finite n` for a fixed natural number `n`. You can use basic facts about arithmetic without proving them. Make sure to follow the proof guidelines given on page 2.

PROPOSED SOLUTION:

```
def Finite.add {n : ℕ} (n1 n2 :Finite n) :
  Finite (n + n) :=
Subtype.mk (Subtype.val n1 + Subtype.val n2)
(by apply add_lt_add
    · simp[Subtype.property n1]
    simp[Subtype.property n2])
```

- 1 point for parameterizing over the variables
- 1 point for the type `Finite (n + n)`
- 1 point for `Subtype.mk`
- 1 point for `Subtype.val n1 + Subtype.val n2`
- 1 point for mentioning the spirit of `add_lt_add`
- 2 points for `by simp [Subtype.property n1]` and `by simp [Subtype.property n2]`