Institute for Informatics of the Ludwig-Maximilians-Universität München
Prof. Dr. Jasmin Blanchette
Xavier Généreux
Yiming Xu

Summer semester 2025
29.07.2025

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 turned off, will be considered a cheating attempt.

Write your full name and matriculation number legibly 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 3–16**. There are 6 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	$\sum$
Points	23	25	17	8	17	10	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) Recall the following simplified typing rules for Lean's dependent type theory:

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

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

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

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

Let $\mathtt{Fin} : \mathbb{N} \to \mathtt{Type}$. Let $\mathtt{a} : \mathbb{N}$, $\mathtt{b} : \mathbb{N}$, $\mathtt{f} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, and $\mathtt{g} : (\mathtt{x} : \mathbb{N}) \to \mathbb{N} \to \mathtt{Fin}\ \mathtt{x}$ 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) $\mathtt{g\ a\ b}$ (7 points)

PROPOSED SOLUTION: The type is $\mathtt{Fin}\ \mathtt{a}$. The typing derivation is

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathtt{g} : (\mathtt{x} : \mathbb{N}) \to \mathbb{N} \to \mathtt{Fin}\ \mathtt{x}}\ \textsc{Cst} \quad \dfrac{}{\vdash \mathtt{a} : \mathbb{N}}\ \textsc{Cst}}{\vdash \mathtt{g}\ \mathtt{a} : \mathbb{N} \to \mathtt{Fin}\ \mathtt{a}}\ \textsc{App}' \quad \dfrac{}{\vdash \mathtt{b} : \mathbb{N}}\ \textsc{Cst}}{\vdash \mathtt{g}\ \mathtt{a}\ \mathtt{b} : \mathtt{Fin}\ \mathtt{a}}\ \textsc{App}'$$

- 2 points for type
- 6 points for derivation tree

(ii) $\mathtt{f\ a\ (}\mathsf{fun}\ \mathtt{x} \mapsto \mathtt{x)}$ (7 points)

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

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathtt{f} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}}\ \textsc{Cst} \quad \dfrac{}{\vdash \mathtt{a} : \mathbb{N}}\ \textsc{Cst}}{\vdash \mathtt{f}\ \mathtt{a} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}}\ \textsc{App}' \quad \dfrac{\dfrac{}{\mathtt{x} : \mathbb{N} \vdash \mathtt{x} : \mathbb{N}}\ \textsc{Var}}{\vdash \mathsf{fun}\ \mathtt{x} \mapsto \mathtt{x} : \mathbb{N} \to \mathbb{N}}\ \textsc{Fun}'}{\vdash \mathtt{f}\ \mathtt{a}\ (\mathsf{fun}\ \mathtt{x} \mapsto \mathtt{x}) : \mathbb{N}}\ \textsc{App}'$$

- 2 points for type
- 5 points for derivation tree

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

(i) $\alpha \to \alpha \to \alpha$ (3 points)

PROPOSED SOLUTION: `fun a _ ↦ a`

- 1 point for `fun a _`
- 2 points for `a`

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

PROPOSED SOLUTION: `fun f g a ↦ g (f a)`

- 1 point for `fun f g a`
- 2 points for `g (f a)`

(iii) $((\beta \to \beta) \to \alpha \to \gamma) \to \beta \to \alpha \to \gamma$ (3 points)

PROPOSED SOLUTION: `fun f b ↦ f (fun _ ↦ b)`
(or `fun f b a ↦ f (fun _ ↦ b) a`)

- 1 point for `fun f b` (or `fun f b a`)
- 2 points for `f (fun _ ↦ b)` (or `f (fun _ ↦ b) a`)

Solution to Question 2 (Functional Programming): **(25 points)**

a) Consider the following Lean function definition:

```
def filter {α : Type} (p : α → Bool) : List α → List α
  | []      => []
  | a :: as =>
    match p a with
    | true  => a :: filter p as
    | false => filter p as
```

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

```
theorem filter_true {α : Type} (xs : List α) :
    filter (fun _ ↦ true) xs = xs
```

PROPOSED SOLUTION: The proof is by structural induction on `xs`.
The base case is

`filter (fun _ ↦ true) [] = []`

Both sides simplify to `[]` and are hence equal.
The induction step is

`filter (fun _ ↦ true) (x :: xs') = x :: xs'`

The induction hypothesis is

`filter (fun _ ↦ true) xs' = xs'`

The induction step simplifies to

`x :: filter (fun _ ↦ true) xs' = x :: xs'`

By the induction hypothesis, the two sides are equal.

- 1 point for "by (structural) induction"
- 1 point for "on `xs`"
- 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)

(ii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 2. (9 points)

```
theorem filter_append {α : Type} (p : α → Bool) (xs ys : List α) :
    filter p (xs ++ ys) = filter p xs ++ filter p ys
```

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

The base case is

`filter p ([] ++ ys) = filter p [] ++ filter p ys`

Both sides simplify to `filter p ys` and are hence equal.

The induction step is

`filter p ((x :: xs') ++ ys) = filter p (x :: xs') ++ filter p ys`

which can be rewritten to `filter p (x :: (xs' ++ ys)) = filter p (x :: xs') ++ filter p ys` The induction hypothesis is

`filter p (xs' ++ ys) = filter p xs' ++ filter p ys`

If `p x` is `true`, then the rewritten induction step simplifies to `x :: filter p (xs' ++ ys) = x :: (filter p xs' ++ filter p ys)` By the induction hypothesis, the two sides are equal. If `p x` is `false`, then the rewritten induction step simplifies to `filter p (xs' ++ ys) = filter p xs' ++ filter p ys` By the induction hypothesis, the two sides are equal.

- 1 point for "by (structural) induction"
- 1 point for "on `xs`"
- 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
- 3 points for proof of induction step (cases, IH, IH)

b) Define a polymorphic Lean function `join` that concatenates a list of lists. For example, `join [[10], [27, 4]]` should evaluate to `[10, 27, 4]`. (8 points)

PROPOSED SOLUTION:

```
def join {α : Type} : List (List α) → List α
  | []        => []
  | xs :: xss => xs ++ join xss
```

- 1 point for "`def join {α : Type}`"
- 1 point for type
- 1 point for LHS of first equation
- 1 point for RHS of first equation
- 1 point for LHS of second equation
- 3 points for RHS of second equation

Solution to Question 3 (Inductive Predicates): (17 points)

a) Consider the following Lean inductive predicate, which determines whether a list consists of elements that repeat themselves in groups of two:

```
inductive IsStuttering {α : Type} : List α → Prop where
  | nil :
    IsStuttering []
  | cons_cons (x : α) {xs : List α} :
    IsStuttering xs → IsStuttering (x :: x :: xs)
```

For example, `IsStuttering [3, 3, 1, 1]` should hold, whereas `IsStuttering [1, 4, 1]` should not hold.

Prove the following Lean theorem about `IsStuttering`. Make sure to follow the proof guidelines given on page 2. (9 points)

```
theorem IsStuttering_map {α β : Type} (f : α → β) {xs : List α}
      (hxs : IsStuttering xs) :
    IsStuttering (List.map f xs)
```

PROPOSED SOLUTION: The proof is by rule induction on `hxs`.

In the `nil` case, the goal is

`IsStuttering (List.map f [])`

This simplifies to `IsStuttering []`, which is provable using `IsStuttering.nil`.

In the `cons_cons` case, the goal is

`IsStuttering xs' → IsStuttering (List.map f (x :: x :: xs'))`

The induction hypothesis is

`IsStuttering (List.map f xs')`

The goal simplifies to

`IsStuttering xs' → IsStuttering (f x :: f x :: List.map f xs'))`

We prove it using `IsStuttering.cons_cons` with the induction hypothesis.

- 1 point for "by (rule) induction"
- 1 point for "on `hxs`"
- 1 point for statement of `nil` case
- 1 point for proof of `nil` case
- 1 point for statement of `cons_cons` case
- 1 point for statement of induction hypothesis
- 3 points for proof of `cons_cons` case (`simp`, `cons_cons`, and IH)

b) Define an inductive predicate `IsReverse` in Lean that takes two values `xs`, `ys` of the polymorphic type `List` α as arguments and that holds if and only if `xs` is the reverse of `ys`. Your answer should not use `List.reverse` or define a helper function. (8 points)

PROPOSED SOLUTION:

```
inductive IsReverse {α : Type} : List α → List α → Prop where
  | nil :
    IsReverse [] []
  | cons (x : α) {xs ys : List α} :
    IsReverse xs ys → IsReverse (x :: xs) (ys ++ [x])
```

- 1 point for "`inductive IsReverse {α : Type}`"
- 1 point for type
- 1 point for LHS of first introduction rule
- 1 point for RHS of first introduction rule
- 1 point for LHS of second introduction rule
- 3 points for RHS of second introduction rule

Solution to Question 4 (Effectful Programming): **(8 points)**

The *random* monad is a monad that threads through a random seed in addition to encapsulating a value of type α. It is reminiscent of the state monad, if we take the random seed as the state. In Lean, the random monad can be defined as follows:

```
def Random (α : Type) : Type :=
  ℕ → α × ℕ

def Random.pure {α : Type} (a : α) : Random α
  | n => (a, n)

def Random.bind {α β : Type} (ma : Random α)
      (f : α → Random β) :
    Random β
  | n =>
    match ma n with
    | (a, n') => f a n'

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

instance Random.Bind : Bind Random :=
  { bind := Random.bind }

def Random.nextRandom : Random ℕ
  | n =>
    let n' := (32212254719 * n + 2833419889721787128217599) % (2 ^ 32 - 1)
    (n', n')
```

a) In addition to `pure`, `bind`, and `nextRandom`, the random monad should allow its user to set the random seed. Implement the following Lean function accordingly: (2 points)

```
def Random.setSeed (n : ℕ): Random Unit
```

PROPOSED SOLUTION:

```
def Random.setSeed (n : ℕ): Random Unit
  | _ => ((), n)
```

- 1 point for `| _ =>`
- 1 point for `((), n)`

b) Prove the following law about random monads. 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)

```
theorem Random.pure_bind_ext {α β : Type} (a : α) (n : ℕ)
      (f : α → Random β) :
    (pure a >>= f) n = f a n
```

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

```
calc (pure a >>= f) n
    = match Random.pure a n with
      | (a', n') => f a' n' :=
    by rfl
  _ = match (a, n) with
      | (a', n') => f a' n' :=
    by rfl
  _ = f a n :=
    by rfl
```

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

Solution to Question 5 (Operational Semantics): (17 points)

The FOR programming language is similar to the familiar WHILE language, with two differences. The first difference is that the `while`–`do` statement is replaced by a `for`–`do` statement, with the concrete syntax

$$\texttt{for}\ x\ \texttt{:=}\ \mathit{lower}\ \texttt{..}\ \mathit{upper}\ \texttt{do}\ \mathit{body}$$

The loop body is executed for $x = \mathit{lower}$, $x = \mathit{lower} + 1$, …, $x = \mathit{upper}$. In a well-formed program, the loop body may refer to the iteration variable x but is not allowed to modify it.

The second difference with WHILE is that the `if`–`then`–`else` statement is replaced by `if`–`then`, with no `else` branch.

For example, the FOR program

```
for i := 5 .. 10 do
    if i < 8 then
        j := j + 1
```

is equivalent to the WHILE program

```
i := 5
while i ≤ 10 do
    if i < 8 then
        j := j + 1
    i := i + 1
```

In Lean, the FOR 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
  | forDo  : String → ℕ → ℕ → Stmt → Stmt

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

a) Complete the following specification of a small-step semantics for FOR in Lean by giving the missing derivation rules for assignment (`:=`) and `for–do`. For the semantics of the latter, you may assume that the index variable is not modified inside the loop body; in other words, you may give an arbitrary semantics to ill-formed programs. (10 points)

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

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

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

PROPOSED SOLUTION:

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

$$\frac{}{\begin{array}{l}\texttt{(for x := low .. high do S, s)} \Rightarrow \\ \qquad \texttt{(if low} \leq \texttt{high then (x := low; S; for x := low + 1 .. high do S), s)}\end{array}}\ \textsc{For}$$

- 4 points for Assign
  - 2 points for LHS
  - 2 points for RHS
- 6 points for For
  - 2 points for LHS
  - 4 points for RHS

b) Encode the rules Seq-Step, Seq-Skip, If-True, and If-False of subquestion a) above in the Lean definition of an inductive predicate. You are not asked to provide any rules for assignment (:=) or `for–do`. (7 points)

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

PROPOSED SOLUTION:

```
| 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)
| 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)
```

- 2 points for `seq_step`
  - 1 point for rule name, variables, and premise
  - 1 point for conclusion
- 1 point for `seq_skip`
- 2 points for `if_true`
  - 1 point for rule name, variables, and condition
  - 1 point for conclusion
- 2 points for `if_false`
  - 1 point for rule name, variables, and condition
  - 1 point for conclusion

Solution to Question 6 (Foundations): **(10 points)**

a) Let σ : `Type 11` and τ : `Type 22` be Lean types. Give the type of each of the following Lean terms.

(5 points)

```
[(0 : ℕ)]
fun (_ : σ) ↦ (5 : ℕ)
fun (x : τ) ↦ x
Sort 5
fun α : Type ↦ List α
```

PROPOSED SOLUTION:

`List` $\mathbb{N}$

$\sigma \to \mathbb{N}$

$\tau \to \tau$

`Sort 6` (or `Type 5`)

`Type` $\to$ `Type` (or `Sort 1` $\to$ `Sort 1`)

- 1 point per type

b) *Vectors* of size n over a type α can be defined as the subtype of all lists of length n over α:

```
def Vector (α : Type) (n : ℕ) : Type :=
  {xs : List α // List.length xs = n}
```

Appending two vectors of respective sizes m and n yields a vector of size m + n:

```
def Vector.append {α : Type} {m n : ℕ} (v : Vector α m) (w : Vector α n) :
    Vector α (m + n) :=
  Subtype.mk (Subtype.val v ++ Subtype.val w)
    (by simp [Subtype.property v, Subtype.property w])
```

Inspired by the definition of `Vector.append`, define the reverse operation on vectors. You may use `List.reverse` in your implementation. (5 points)

PROPOSED SOLUTION:

```
def Vector.reverse {α : Type} {n : ℕ} (v : Vector α n) : Vector α n :=
  Subtype.mk (List.reverse (Subtype.val v))
    (by simp [Subtype.property v])
```

- 1 point for `Subtype.mk`
- 2 points for `List.reverse (Subtype.val v)`
- 1 point for `by simp [Subtype.property v]`