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

Summer semester 2025
01.10.2025

# Retake 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.

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

**a)** Recall the following simplified typing rules for Lean's dependent type theory:

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

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

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

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

Let `Fin` : $\mathbb{N} \to$ `Type`, and let `a` : $\mathbb{N}$, `f` : $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, `g` : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, and `h` : $(\texttt{x} : \mathbb{N}) \to$ `Fin (x + 5)` 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) `f (fun x ↦ g x a)`

(ii) `h a`

**b)** Let $\alpha$, $\beta$, and $\gamma$ be Lean types. Give an inhabitant for each of the following types:

(i) $\beta \to \alpha \to \alpha$

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

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

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

**a)** Consider the following Lean function definition:

```
def replaceAll (bef aft : ℕ) : List ℕ → List ℕ
  | []      => []
  | n :: ns => (if n = bef then aft else n) :: replaceAll bef aft ns
```

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

```
theorem length_replaceAll (bef aft : ℕ) (ns : List ℕ) :
    List.length (replaceAll bef aft ns) = List.length ns
```

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

```
theorem append_replaceAll (bef aft : ℕ) (ms ns : List ℕ) :
    replaceAll bef aft ms ++ replaceAll bef aft ns =
    replaceAll bef aft (ms ++ ns)
```

**b)** Define a polymorphic Lean function `takeWhile` that takes a predicate `p :` $\alpha \to$ `Bool` and a list `xs : List` $\alpha$ and that returns the longest prefix of `xs` consisting of elements for which `p` is true. If `xs` is empty or `p` is false for `xs`'s first element, then `[]` is returned. For example, `takeWhile isPrime [2, 31, 7, 4, 5]` should evaluate to `[2, 31, 7]`, where `isPrime` tests for primality.

**Question 3 (Inductive Predicates): (17 points)**

**a)** Consider the following Lean inductive predicate, which determines whether a list `xs` is a subsequence of another list `ys`:

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

A subsequence of a list `ys` is a list `xs` that can be derived from `ys` by deleting zero or more elements while keeping the order of the remaining elements. For example, `IsSubseq [5, 3] [1, 5, 2, 3]` holds, whereas `IsSubseq [3, 5] [1, 5, 2, 3]` does not hold.

Prove the following Lean theorem about `IsSubseq`. Make sure to follow the proof guidelines given on page 2.

```
theorem IsSubseq_map {α β : Type} (f : α → β) {xs ys : List α}
      (hxs : IsSubseq xs ys) :
    IsSubseq (List.map f xs) (List.map f ys)
```

**b)** Define an inductive predicate `IsSublist` in Lean that takes two lists `xs`, `ys` of the polymorphic type `List` $\alpha$ as arguments and that holds if and only if `xs` occurs as a (consecutive) sublist of `ys`. For example, `IsSublist [3, 5] [1, 3, 5, 7]` should hold, whereas `IsSublist [3, 5] [1, 3, 36, 5, 7]` should not hold.

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

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

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

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

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

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

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

**a)** In addition to `pure` and `bind`, the serial monad should allow its user to access the serial number. The `nextSerial` operation should return the current value of the serial number and increment the stored serial number by one. Implement the following Lean function accordingly.

```
def Serial.nextSerial : Serial ℕ
```

**b)** Prove the following law about serial 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.

```
theorem Serial.bind_pure_ext {α : Type} (ma : Serial α) (n : ℕ) :
    (ma >>= pure) n = ma n :=
```

**Question 5 (Operational Semantics): (17 points)**

The REPEAT 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 `repeat`–`do` statement, with the concrete syntax

`repeat` $n$ `do` *body*

where $n$ is a literal natural number. The loop body is executed exactly $n$ times.

The second difference with WHILE is that the `if`–`then`–`else` statement is replaced by `unless`–`do`, with the concrete syntax

`unless` *condition* `do` *body*

The loop body is executed only if the given condiiton evaluates to false.

For example, the REPEAT program

```
repeat 2 do
    unless j > i do
        j := j + 1
```

is equivalent to the WHILE program

```
if j > i then
    skip
else
    j := j + 1;
if j > i then
    skip
else
    j := j + 1
```

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

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

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

**a)** Complete the following specification of a small-step semantics for REPEAT in Lean by giving the missing derivation rules for assignment (`:=`) and `repeat–do`.

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

$$\frac{}{\texttt{(unless B do S, s)} \Rightarrow \texttt{(skip, s)}}\ \textsc{Unless-True} \quad \text{if } \texttt{s(B)} \text{ is true}$$

$$\frac{}{\texttt{(unless B do S, s)} \Rightarrow \texttt{(S, s)}}\ \textsc{Unless-False} \quad \text{if } \texttt{s(B)} \text{ is false}$$

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

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

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

**a)** Let $\sigma$ : `Type 5` and $\tau$ : `Type 8` be Lean types. Give the type of each of the following Lean terms.

```
[[(1 : ℕ)]]
fun (n : ℕ) ↦ n + n
fun (x : σ) ↦ x
σ → τ
fun α : Type ↦ α × α
```

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

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

Prepending an element to a vector of size `n` yields a vector of size `n + 1`:

```
def Vector.cons {α : Type} {x : α} {n : ℕ} (v : Vector α n) :
    Vector α (n + 1) :=
  Subtype.mk (x :: Subtype.val v) (by simp [Subtype.property v])
```

Inspired by the definition of `Vector.cons`, define the operation “snoc” on vectors, which appends an element to a vector.