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

Summer semester 2025
29.07.2025

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.

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

(ii) `f a (fun x ↦ x)`

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

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

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

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

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.

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

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

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

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]`.

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.

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

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.

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:

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

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.

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

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

`for` x `:=` *lower* `..` *upper* `do` *body*

The loop body is executed for $x = lower$, $x = lower + 1$, ..., $x = upper$. 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.

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

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

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

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`.

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

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.

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

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.