Institute for Informatics of the Ludwig-Maximilians-Universität München | Summer semester 2024
Prof. Dr. Jasmin Blanchette | 09.10.2024
Xavier Généreux
Balazs Toth

Second Examination in the Course **Interactive Theorem Proving**

You have **120 minutes** at your disposal. Written or electronic aids are not permitted. Carrying electronic devices, even turned off, will be considered cheating.

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 1**. Questions can be found on **pages 2–15**. There are 6 questions for a total of 100 points.

You may use the back of the sheets for auxiliary calculations. If you use the back for actual answers, 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 in sufficiently good health at the beginning of the examination and that you accept this examination bindingly.

Last name:

First name:

Matriculation number:

Program of study:

☐ **Please check with an X *only* if the exam should be voided and not graded.**
Bitte *nur* ankreuzen, wenn die Klausur entwertet und nicht korrigiert werden soll.

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	20	25	25	8	12	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 lemmas 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): **(20 points)**

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

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

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

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

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

Let `Vector` : $(\alpha$: `Type`$) \to \mathbb{N} \to$ `Type` be a type constructor. Let `a` : $\mathbb{N}$, `g` : $\mathbb{N} \to \mathbb{N} \to \mathbb{Q}$, and `f` : (`x` : $\mathbb{N}$) $\to$ `Vector` $\mathbb{N}$ `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) `f a`

(ii) fun `x` $\mapsto$ `g x x`

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

- $\alpha \to \mathbb{N}$

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

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

Question 2 (Functional Programming): (25 points)

a) Consider the following Lean function definition:

```
def stutter {α : Type} : List α → List α
  | []      => []
  | a :: as => a :: a :: stutter as
```

(i) Give the value of `stutter [1, 2, 3]`. (There is no need to provide intermediate steps.)

(ii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1.

```
theorem stutter_append {α : Type} (xs ys : List α) :
  stutter (xs ++ ys) = stutter xs ++ stutter ys
```

(iii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1.

```
theorem length_stutter {α : Type} (xs : List α) :
  List.length (stutter xs) = 2 * List.length xs
```

b) Define a Lean function `separate` that takes a term and a list and that returns a list where the term is inserted between each pair of adjacent elements of the list. For example, `separate 0 [1, 2, 3]` should evaluate to `[1, 0, 2, 0, 3]`.

Question 3 (Inductive Predicates): (25 points)

a) A binary tree is *linear* if all its nodes have at most one child. Consider the following Lean inductive predicate, which determines whether a binary tree is linear:

```
inductive Linear {α : Type} : Tree α → Prop
  | nil :
    Linear Tree.nil
  | left (a : α) (l : Tree α) (hl : Linear l) :
    Linear (Tree.node a l Tree.nil)
  | right (a : α) (r : Tree α) (hr : Linear r) :
    Linear (Tree.node a Tree.nil r)
```

Recall the `mirror` function on binary trees:

```
def mirror {α : Type} : Tree α → Tree α
  | Tree.nil        => Tree.nil
  | Tree.node a l r => Tree.node a (mirror r) (mirror l)
```

Prove the following theorem about the interaction between `Linear` and `mirror`. Make sure to follow the proof guidelines given on page 1.

```
theorem Linear_mirror {α : Type} (t : Tree α) (ht : Linear t) :
  Linear (mirror t)
```

b) Define an inductive predicate `Prefix` in Lean that takes two lists over a polymorphic type α as arguments and that holds when the first list is a prefix of the second. For exemple, `Prefix [1, 2] [1, 2]` and `Prefix [1, 2] [1, 2, 4]` should hold.

c) Define an inductive predicate `ListOver` in Lean that takes a set over a polymorphic type α and a list over α as arguments and that holds if all the elements of the list belong to the set. For example, `ListOver {4, 7} [7, 4, 7]` should hold, whereas `ListOver {7} [6]` should not hold. You may use the predicate $\in$ to test for set membership.

Question 4 (Effectful Programming): (8 points)

The writer monad is a monad that stores a list of messages to be output to the console in addition to a value of type α. In Lean, the writer monad could be defined as follows:

```
def Writer (α : Type) := α × List String

def Writer.pure {α : Type} (a : α) : Writer α :=
  (a, [])

def Writer.bind {α β : Type} (w : Writer α) (f : α → Writer β) : Writer β :=
  match w with
  | (a, msgs) =>
    match f a with
    | (b, msgs') => (b, msgs ++ msgs')

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

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

In the definition of `Writer.bind`, the purpose of `match` is to unfold the `Writer` tuple. It can be useful to have an equivalent representation using `Prod.fst` and `Prod.snd`:

```
lemma Writer.bind_prod {α β : Type} (w : Writer α) (f : α → Writer β) :
  w >>= f = (Prod.fst (f (Prod.fst w)), Prod.snd w ++ Prod.snd (f (Prod.fst w))) :=
  by rfl
```

a) In addition to `pure` and `bind`, the writer monad should offer the possibility to output messages and to retrieve the list of output messages. Implement the following two Lean functions:

```
def Writer.getOutput {α : Type} (w : Writer α) : List String :=
```

```
def Writer.output {α : Type} (w : Writer α) (msg : String) : Writer α :=
```

b) Prove the following law about writer monads. Make sure to follow the proof guidelines given on page 1. In addition, give detailed steps when unfolding the definition of monad operators.

Hint: The `Writer.bind_prod` lemma presented above may be useful.

```
theorem pure_bind {α β : Type} (a : α) (f : α → Writer β) :
  (pure a >>= f) = f a :=
```

Question 5 (Denotational Semantics): (12 points)

The IF language consists of all the same constructs as the WHILE language except that it does not include a `while` loop:

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

The denotational semantics of the IF language is given below:

```
def denote : Stmt → Set (State × State)
  | Stmt.skip             => Id
  | Stmt.assign x a       =>
    {st | Prod.snd st = (Prod.fst st)[x ↦ a (Prod.fst st)]}
  | Stmt.seq S T          => denote S ○ denote T
  | Stmt.ifThenElse B S T => (denote S ⇃ B) ∪ (denote T ⇃ (fun s ↦ ¬ B s))
```

where

```
Id      := {ab | Prod.snd ab = Prod.fst ab}
r ⇃ P   := {ab | ab ∈ r ∧ P (Prod.fst ab)}
r₁ ○ r₂ := {ac | ∃b, (Prod.fst ac, b) ∈ r₁ ∧ (b, Prod.snd ac) ∈ r₂}
```

We also define ⟦ ⟧ as syntactic sugar for `denote`.

a) Prove the following equality:

```
⟦Stmt.ifThenElse (fun s ↦ s "x" > 0)
   (Stmt.assign "x" (fun s ↦ s "x" + 1))
   Stmt.skip⟧
=
⟦Stmt.ifThenElse (fun s ↦ s "x" = 0)
   Stmt.skip
   (Stmt.assign "x" (fun s ↦ s "x" + 1))⟧
```

Recall that program variables range over natural numbers.

b) The `denote` function uses tuples in `Set (State × State)`. It is called a *relational* denotational semantics, and the tuples relate prestates to poststates. Instead, since the IF language is deterministic and terminating, we could have defined the semantics as a function from prestates to poststates. Complete the following definition of a *functional* version of `denote`. (The definition uses the `Classical.propDecidable` attribute to enable noncomputable decidability on every `Prop`.)

```
attribute [instance 0] Classical.propDecidable

noncomputable def denote : Stmt → State → State
| Stmt.skip             => fun s ↦ s
| Stmt.assign x a       =>
| Stmt.seq S T          =>
| Stmt.ifThenElse B S T =>
```

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

a) Let σ `: Sort 0` and τ `: Type` be Lean types. Give the type of each of the following Lean terms.

```
fun x : σ ↦ x
τ → σ
Sort 4
σ → τ
```

b) We call a `NonUnitalSemiring` a type with addition, multiplication, and a 0 element and where addition is commutative and associative, multiplication is associative and left and right distributive over addition, and 0 is the additive identity.

Complete the following `class` declaration of `NonUnitalSemiring`:

```
universe u

class NonUnitalSemiring (α : Type u) : Type u where
  add            : α → α → α
  mul            : α → α → α
  zero           :
  mul_assoc      : ∀a b c, mul (mul a b) c = mul a (mul b c)
  add_assoc      :
  left_distrib   : ∀a b c, mul a (add b c) = add (mul a b) (mul a c)
  right_distrib  :
  zero_add       : ∀a, add zero a = a
  add_zero       : ∀a, add a zero = a
  zero_mul       :
  mul_zero       :
  add_comm       :
```