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

Summer semester 2024
14.10.2024

Solution to the Third 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 2–14**. 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.

Solution to Question 1 (Types and Terms): **(20 points)**

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

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

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

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

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

Let $\mathtt{a} : \mathbb{N}$, $\mathtt{f} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$, $\mathtt{g} : \mathbb{N} \to \mathbb{N}$, and $\mathtt{h} : (\mathtt{y} : \mathbb{N}) \to \{\mathtt{x} : \mathbb{N}\ //\ \mathtt{x} < 5\}$ 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) $\mathtt{h}\ \mathtt{a}$ (6 points)

PROPOSED SOLUTION: The type is $\{\mathtt{x} : \mathbb{N}\ //\ \mathtt{x} < 5\}$. The typing derivation is

$$\frac{\dfrac{}{\vdash \mathtt{h} : (\mathtt{y} : \mathbb{N}) \to \{\mathtt{x} : \mathbb{N}\ //\ \mathtt{x} < 5\}}\ \textsc{Cst} \qquad \dfrac{}{\vdash \mathtt{a} : \mathbb{N}}\ \textsc{Cst}}{\vdash \mathtt{h}\ \mathtt{a} : \{\mathtt{x} : \mathbb{N}\ //\ \mathtt{x} < 5\}}\ \textsc{App}'$$

- 2 points for the type
- 4 points for the derivation tree

(ii) $\mathtt{fun}\ \mathtt{x} \mapsto \mathtt{f}\ \mathtt{g}\ \mathtt{x}$ (8 points)

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

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathbb{N} \vdash \mathtt{f} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}\ \textsc{Cst} \qquad \dfrac{}{\mathtt{x} : \mathbb{N} \vdash \mathtt{g} : \mathbb{N} \to \mathbb{N}}\ \textsc{Cst}}{\mathtt{x} : \mathbb{N} \vdash \mathtt{f}\ \mathtt{g} : \mathbb{N} \to \mathbb{N}}\ \textsc{App}' \qquad \dfrac{}{\mathtt{x} : \mathbb{N} \vdash \mathtt{x} : \mathbb{N}}\ \textsc{Var}}{\mathtt{x} : \mathbb{N} \vdash \mathtt{f}\ \mathtt{g}\ \mathtt{x} : \mathbb{N}}\ \textsc{App}'}{\vdash \mathtt{fun}\ \mathtt{x} \mapsto \mathtt{f}\ \mathtt{g}\ \mathtt{x} : \mathbb{N} \to \mathbb{N}}\ \textsc{Fun}'$$

- 2 points for the type
- 6 points for the derivation tree

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

- $\alpha \to \beta \to \beta$ (2 points)

PROPOSED SOLUTION: `fun a b ↦ b`

- 1 point for `fun a b ↦`
- 1 point for `b`

- `(`$\alpha \to \beta$`)` $\to$ `(`$\alpha \to \alpha \to \alpha$`)` $\to \alpha \to \beta$ (2 points)

PROPOSED SOLUTION: `fun g f a ↦ g a`

- 1 point for `fun g f a`
- 1 point for `g a`

- `((`$\alpha \to \alpha \to \beta$`)` $\to \alpha$`)` $\to \gamma \to \beta \to \alpha$ (2 points)

PROPOSED SOLUTION: `fun h c b ↦ h (fun a a' ↦ b)`

- 1 point for `fun h c b`
- 1 point for `h (fun a a' ↦ b)`

Solution to 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 [4, 3, 2]`. (There is no need to provide intermediate steps.) (2 points)

PROPOSED SOLUTION: `[4, 4, 3, 3, 2, 2]`.

2 points for right answer, 0 otherwise

(ii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1. (8 points)

```
theorem map_stutter {α β : Type} (f : α → β) (ys : List α) :
  List.map f (stutter ys) = stutter (List.map f ys) :=
```

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

`List.map f (stutter []) = stutter (List.map f [])`

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

`List.map f (stutter (y :: ys')) = stutter (List.map f (y :: ys'))`

The induction hypothesis is

`List.map f (stutter ys') = stutter (List.map f ys')`

The induction step simplifies to

`[f y, f y] ++ List.map f (stutter ys') = [f y, f y] ++ stutter (List.map f ys')`

By the induction hypothesis, the two sides are equal.

- 1 point for "by (structural) induction"
- 1 point for "on `ys`"
- 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 (IH and `simp`)

(iii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1. (8 points)

```
theorem stutter_snoc {α : Type} (xs : List α) (y : α) :
  stutter (xs ++ [y]) = stutter xs ++ [y, y] :=
```

PROPOSED SOLUTION:

The proof is by structural induction on `xs`.

The base case is

`stutter ([] ++ [y]) = stutter [] ++ [y, y]`

Both sides simplify to `[y, y]` and are hence equal.

The induction step is

`stutter ((x :: xs') ++ [y]) = stutter (x :: xs') ++ [y, y]`

The induction hypothesis is

`stutter (xs' ++ [y]) = stutter xs' ++ [y, y]`

The induction step simplifies to

`[x, x] ++ stutter (xs' ++ [y]) = [x, x] ++ stutter xs' ++ [y, y]`

(up to associativity of ++). 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 (IH and `simp`)

b) Define a Lean function `singletonify` that takes a list `[`$x_1, \ldots, x_n$`]` and that returns a list of singletons `[[`x_1`]`$, \ldots,$ `[`x_n`]]`. For example, `singletonify [1, 2, 3, 5, 7]` should evaluate to `[[1], [2], [3], [5], [7]]`. (7 points)

PROPOSED SOLUTION:

```
def singletonify {α : Type} : List α → List (List α)
  | []      => []
  | x :: xs => [x] :: singletonify xs
```

- 1 point for "`def singletonify {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
- 2 points for RHS of second equation

Solution to Question 3 (Inductive Predicates): **(25 points)**

a) Recall the `stutter` function from Question 2:

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

Now consider the following Lean inductive predicate, which holds when a list has even length:

```
inductive EvenLength {α : Type} : List α → Prop where
  | nil :
    EvenLength []
  | add_two (x y : α) {xs : List α} :
    EvenLength xs → EvenLength (x :: y :: xs)
```

For example, `EvenLength [1, 2]` holds, whereas `EvenLength [3, 4, 5]` does not hold.

Prove the following Lean theorem about `EvenLength`. Make sure to follow the proof guidelines given on page 1. (10 points)

```
theorem EvenLength_stutter {α β : Type} (xs : List α)
    (hxs : EvenLength xs) :
  EvenLength (stutter xs)
```

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

In the `nil` case, the goal is

`EvenLength (stutter [])`

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

In the `add_two` case, the goal is

`EvenLength xs' → EvenLength (stutter (x :: y :: xs'))`

The induction hypothesis is

`EvenLength (stutter xs')`

The goal simplifies to

`EvenLength xs' → EvenLength (x :: x :: y :: y :: stutter xs')`

We prove it using `EvenLength.add_two` *twice* with the induction hypothesis.

- 1 point for “by (rule) induction”
- 1 point for “on `hxs`”
- 1 point for statement of `singleton` case
- 1 point for proof of `singleton` case
- 1 point for statement of `add_two` case
- 1 point for statement of induction hypothesis
- 4 points for proof of `add_two` case (`simp`, `add_two`, `add_two`, and IH)

b) Define an inductive predicate `Suffix` in Lean that takes two lists over a polymorphic type α as arguments and that holds when the first list is a suffix of the second. For exemple, `Suffix [1, 2] [1, 2]` and `Suffix [2, 4] [1, 2, 4]` should hold. (8 points)

PROPOSED SOLUTION:

```
inductive Suffix {α : Type} : List α → List α → Prop where
  | nil (as : List α) :
    Suffix [] as
  | snoc (a : α) (as bs : List α) (hp : Suffix as bs) :
    Suffix (as ++ [a]) (bs ++ [a])
```

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

c) Define an inductive predicate `EvenPalindrome` in Lean that takes a list over a polymorphic type α as argument and that holds if the list is a palindrome (i.e., if it equals its reverse) and has even length. For example, `EvenPalindrome [1, 2, 2, 1]` should hold, whereas `EvenPalindrome [1, 3, 1]` and `EvenPalindrome [1, 3, 7]` should not hold. (8 points)

PROPOSED SOLUTION:

```
inductive EvenPalindrome {α : Type} : List α → Prop where
  | nil :
    EvenPalindrome []
  | sandwich (x : α) (xs : List α) (hxs : EvenPalindrome xs) :
    EvenPalindrome ([x] ++ xs ++ [x])
```

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

Solution to Question 4 (Metaprogramming): (8 points)

Consider the following custom Lean tactic:

```
macro "enigma" : tactic =>
  `(tactic| repeat' first
              | assumption
              | intro _
              | apply True.intro
              | apply And.intro
              | apply Iff.intro)
```

a) Briefly explain what the `enigma` tactic does. You may assume that we already know what `assumption`, `intro`, and `apply` does. (3 points)

PROPOSED SOLUTION: The tactic repeatly tries to apply the first applicable tactic among five tactics: `assumption`, `intro _`, `apply True.intro`, `apply And.intro`, and `apply Iff.intro`. By "repeatedly," we mean that the process is repeated for all goals and recursively for all emerging subgoals.

- 1 point for general explanation
- 1 point for `repeat'`'s explanation
- 1 point for `first`'s explanation

b) In the following Lean code fragment, the `enigma` tactic is applied to transform the goal:

```
theorem abbatf (a b : Prop) :
  a → b → b ∧ a ∧ True ∧ False :=
  by
    enigma
```

The proof state before invoking `enigma` is

```
a b : Prop
⊢ a → b → b ∧ a ∧ True ∧ False
```

Give the proof state *after* invoking `enigma`. Make sure to include all subgoals. (5 points)

PROPOSED SOLUTION:

```
a b : Prop
a_1 : a
a_2 : b
⊢ False
```

- 3 points for `a` and `b` hypotheses
- 2 points for `False`

Solution to Question 5 (Operational Semantics): **(12 points)**

The IF0 programming language is similar to WHILE, with two differences. First, the `while–do` statement is omitted. Second, the `if–then–else` statement is replaced by `if_zero–then–else`. For example, the program

```
if_zero m * n then
  x := 0
else
  y := 1
```

executes `x := 0` if the condition `m * n = 0` holds when entering the construct; otherwise, it executes `y := 1`.

In Lean, IF0 is modeled by the following inductive type:

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

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

a) Complete the following specification of a small-step semantics for IF0 in Lean by giving the missing derivation rules for `seq` and `if_zero`. (6 points)

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

$$\frac{}{\texttt{(Stmt.skip; T, s)} \Rightarrow \texttt{(T, s)}}\ \textsc{SeqSkip}$$

PROPOSED SOLUTION:

$$\frac{\texttt{(S, s)} \Rightarrow \texttt{(S', s')}}{\texttt{(S; T, s)} \Rightarrow \texttt{(S'; T, s')}}\ \textsc{SeqStep}$$

$$\frac{}{\texttt{(if\_zero a then S else T, s)} \Rightarrow \texttt{(S, s)}}\ \textsc{IfZeroTrue} \quad \text{if } \texttt{s(a)} = \texttt{0}$$

$$\frac{}{\texttt{(if\_zero a then S else T, s)} \Rightarrow \texttt{(T, s)}}\ \textsc{IfZeroFalse} \quad \text{if } \texttt{s(a)} \neq \texttt{0}$$

- 2 points for SeqStep
  - 1 point for premises
  - 1 point for conclusion
- 2 points for IfZeroTrue
  - 1 point for side condition
  - 1 point for conclusion
- 2 points for IfZeroFalse
  - 1 point for side condition
  - 1 point for conclusion

b) Complete the following Lean definition of an inductive predicate that encodes the small-step semantics you specified in subquestion a) above. (6 points)

```
inductive SmallStep : Stmt × State → Stmt × State → Prop where
  | assign (x a s) :
    SmallStep (Stmt.assign x a, s) (Stmt.skip, s[x ↦ a s])
  | seq_skip (T s) :
    SmallStep (Stmt.skip; T, s) (T, s)
```

PROPOSED SOLUTION:

```
  | seq_step (S S' T s s') (hS : SmallStep (S, s) (S', s')) :
    SmallStep (S; T, s) (S'; T, s')
  | ifZero_true (a S T s) (hcond : a s = 0) :
    SmallStep (Stmt.ifZero a S T, s) (S, s)
  | ifZero_false (a S T s) (hcond : a s ≠ 0) :
    SmallStep (Stmt.ifZero a S T, s) (T, s)
```

- 2 points for `seq_step`
  - 1 point for rule name, variables, and premises
  - 1 point for conclusion
- 2 points for `ifZero_True`
  - 1 point for rule name, variables, and premises
  - 1 point for conclusion
- 2 points for `ifZero_False`
  - 1 point for rule name, variables, and premises
  - 1 point for conclusion
- Folgefehler are acknowledged

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

a) Let σ `: Type 4` and τ `: Type 2` be Lean types. Give the type of each of the following Lean terms. (5 points)

```
fun (x : σ) (y : σ) ↦ x
Sort 5
fun α : Type ↦ List (ℕ × α)
σ → τ
τ → σ
```

PROPOSED SOLUTION:

$\sigma \to \sigma \to \sigma$
`Sort 6` (or `Type 5`)
`Type` $\to$ `Type` (or `Sort 1` $\to$ `Sort 1`)
`Type 4` (or `Sort 5`)
`Type 4` (or `Sort 5`)

- 1 point per type

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            :
  zero           : α
  mul_assoc      :
  add_assoc      : ∀a b c, add (add a b) c = add a (add b c)
  left_distrib   :
  right_distrib  : ∀a b c, mul (add a b) c = add (mul a c) (mul b c)
  zero_add       : ∀a, add zero a = a
  add_zero       :
  zero_mul       :
  mul_zero       : ∀a, mul a zero = zero
  add_comm       : ∀a b, add a b = add b a
```

(5 points)

PROPOSED SOLUTION:

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

- 1 point for `mull`
- 1 point for the `mul_assoc` statement
- 1 point for the `left_distrib` statement
- 1 point for the `add_zero` statement
- 1 point for the `zero_mul` statement