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

Summer semester 2024
25.07.2024

Solution to the Final 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 switched 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**. 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.

There are 6 questions for a total of 100 points. The subquestions can be completed independently of each other.

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:

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

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

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

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

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

PROPOSED SOLUTION: The type is $\{\texttt{x} : \mathbb{N} \mid\mid \texttt{x} < \texttt{a}\}$. The typing derivation is

$$\dfrac{\dfrac{}{\vdash \texttt{h} : (\texttt{y} : \mathbb{N}) \to \{\texttt{x} : \mathbb{N} \mid\mid \texttt{x} < \texttt{y}\}}\ \textsc{Cst} \qquad \dfrac{}{\vdash \texttt{a} : \mathbb{N}}\ \textsc{Cst}}{\vdash \texttt{h a} : \{\texttt{x} : \mathbb{N} \mid\mid \texttt{x} < \texttt{a}\}}\ \textsc{App}'$$

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

(ii) $\texttt{f (fun x} : \mathbb{N} \mapsto \texttt{g x x)}$ (8 points)

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

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

- 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:

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

PROPOSED SOLUTION: `fun a ↦ a`

- 1 point for `fun a`
- 1 point for `a`

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

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

- 1 point for `fun g f c`
- 1 point for `g (f c)`

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

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

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

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

a) Consider the following Lean function definition:

```
def surround {α : Type} (a : α) : List α → List α
  | []      => [a]
  | b :: bs => a :: b :: surround a bs
```

(i) Compute the value of `surround 0 [1, 2, 3]`. (2 points)

PROPOSED SOLUTION: `[0, 1, 0, 2, 0, 3, 0]`.

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_surround {α β : Type} (f : α → β) (x : α) (ys : List α) :
  List.map f (surround x ys) = surround (f x) (List.map f ys)
```

PROPOSED SOLUTION:

The proof is by structural induction on `ys`.

The base case is

```
List.map f (surround x []) = surround (f x) (List.map f [])
```

Both sides simplify to `[f x]` and are hence equal.

The induction step is

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

The induction hypothesis is

```
List.map f (surround x ys') = surround (f x) (List.map f ys')
```

The induction step simplifies to

```
[f x, f y] ++ List.map f (surround x ys')
= [f x, f y] ++ surround (f x) (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 length_surround {α : Type} (x : α) (ys : List α) :
  List.length (surround x ys) = 2 * List.length ys + 1
```

PROPOSED SOLUTION:

The proof is by structural induction on `ys`.

The base case is

`List.length (surround x []) = 2 * List.length [] + 1`

Both sides simplify to `1` and are hence equal.

The induction step is

`List.length (surround x (y :: ys')) = 2 * List.length (y :: ys') + 1`

The induction hypothesis is

`List.length (surround x ys') = 2 * List.length ys' + 1`

The induction step simplifies to

`List.length (surround x ys') + 2 = 2 * List.length ys' + 3`

By the induction hypothesis and some basic arithmetic reasoning, 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 `arith`)

b) Define a Lean function `suffixes` that takes a list and that returns a list of all suffixes of that list, including the list itself, from longest to shortest. For example, `suffixes [1, 2, 3]` should evaluate to `[[1, 2, 3], [2, 3], [3], []]`. (7 points)

PROPOSED SOLUTION:

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

- 1 point for "`def suffixes {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) Consider the following Lean inductive predicate, which determines whether a list has odd length:

```
inductive OddLength {α : Type} : List α → Prop
  | singleton (x : α) :
    OddLength [x]
  | add_two (x y : α) {xs : List α} :
    OddLength xs → OddLength (x :: y :: xs)
```

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

Prove the following Lean theorem about `OddLength`. Make sure to follow the proof guidelines given on page 1. (9 points)

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

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

In the `singleton` case, the goal is

$$\texttt{OddLength (List.map f [x])}$$

This simplifies to `OddLength [f x]`, which is provable using `OddLength.singleton`.

In the `add_two` case, the goal is

$$\texttt{OddLength xs'} \to \texttt{OddLength (List.map f (x :: y :: xs'))}$$

The induction hypothesis is

$$\texttt{OddLength (List.map f xs')}$$

The goal simplifies to

$$\texttt{OddLength xs'} \to \texttt{OddLength (f x :: f y :: List.map f xs'))}$$

We prove it using `OddLength.add_two` 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
- 3 points for proof of `add_two` case (`simp`, `add_two`, and IH)

b) Define an inductive predicate `InList` in Lean that takes a value `x` of the polymorphic type α and a list `ys` over α as arguments and that holds if `x` occurs in `ys`. (7 points)

PROPOSED SOLUTION:

```
inductive InList {α : Type} : α → List α → Prop
  | in_hd (x : α) (xs : List α)   : InList x (x :: xs)
  | in_tl (x y : α) (ys : List α) : InList x ys → InList x (y :: ys)
```

- 1 point for "`inductive InList {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
- 2 points for RHS of second introduction rule

c) Consider the following Lean type of binary trees and a helper function that counts the number of nodes in a given tree:

```
inductive Tree (α : Type) : Type
  | nil  : Tree α
  | node : α → Tree α → Tree α → Tree α

def numNodes {α : Type} : Tree α → ℕ
  | Tree.nil        => 0
  | Tree.node a l r => numNodes l + numNodes r + 1
```

Define an inductive predicate `RightHeavy` in Lean that takes a tree as argument and that holds if the tree enjoys *either* of the following properties:

- The tree is of the form `Tree.nil`.
- The tree is of the form `Tree.node a l r`, the number of nodes in `l` is less than or equal to the number of nodes in `r`, and both `l` and `r` recursively satisfy `RightHeavy`.

(9 points)

PROPOSED SOLUTION:

```
inductive RightHeavy {α : Type} : Tree α → Prop
  | nil :
    RightHeavy Tree.nil
  | node (a : α) (l r : Tree α) :
    numNodes l ≤ numNodes r → RightHeavy l → RightHeavy r →
    RightHeavy (Tree.node a l r)
```

- 1 point for "`inductive RightHeavy {α : Type}`"
- 1 point for type
- 1 point for LHS of first clause
- 1 point for RHS of first clause
- 1 point for LHS of second clause
- 4 points for RHS of second clause

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

Consider the following custom Lean tactic:

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

a) Briefly explain what the `mystery` 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 four tactics: `assumption`, `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 `mystery` tactic is applied to massage the goal:

```
theorem abac (a b c : Prop) :
  a → b ∧ a ∧ ¬ c :=
  by
    mystery
```

The proof state before invoking `mystery` is

```
a b c : Prop
⊢ a → b ∧ a ∧ ¬c
```

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

PROPOSED SOLUTION:

```
a b c : Prop
ha : a
⊢ b

a b c : Prop
ha : a
hc : c
⊢ False
```

- 1 point for first subgoal's hypotheses
- 1 point for first subgoal's target
- 1 point for second subgoal's hypotheses
- 1 point for second subgoal's target
- 1 point for lack of extraneous subgoals (e.g., for `a`)

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 would be modeled by the following datatype:

```
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 big-step semantics for IF0 in Lean by giving the missing derivation rules for `seq` and `if_zero`. (6 points)

$$\frac{}{\texttt{(skip, s)} \Longrightarrow \texttt{s}}\ \textsc{Skip}$$

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

PROPOSED SOLUTION:

$$\frac{\texttt{(S, s)} \Longrightarrow \texttt{t} \quad \texttt{(T, t)} \Longrightarrow \texttt{u}}{\texttt{(S ; T, s)} \Longrightarrow \texttt{u}}\ \textsc{Seq}$$

$$\frac{\texttt{(S, s)} \Longrightarrow \texttt{t}}{\texttt{(if\_zero a then S else T, s)} \Longrightarrow \texttt{t}}\ \textsc{IfZero\_True} \quad \text{if } \texttt{s(a)} = \texttt{0}$$

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

- 2 points for Seq
  - 1 point for premises
  - 1 point for conclusion
- 2 points for IfZero_True
  - 1 point for premise and side condition
  - 1 point for conclusion
- 2 points for IfZero_False
  - 1 point for premise and side condition
  - 1 point for conclusion

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

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

PROPOSED SOLUTION:

```
  | seq (S T s t u) (hS : BigStep (S, s) t)
      (hT : BigStep (T, t) u) :
    BigStep (S; T, s) u
  | ifZero_true (a S T s t) (hcond : a s = 0)
      (hbody : BigStep (S, s) t) :
    BigStep (Stmt.ifZero a S T, s) t
  | ifZero_false (a S T s t) (hcond : a s ≠ 0)
      (hbody : BigStep (T, s) t) :
    BigStep (Stmt.ifZero a S T, s) t
```

- 2 points for `seq`
  - 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 2` and τ `: Type 3` be Lean types. Give the type of each of the following Lean terms. (4 points)

`fun x :` $\sigma \mapsto$ `x`
`Type 5`
$\sigma \to \tau$
`fun` α `: Type` $\mapsto$ `Option (`$\alpha \times \alpha$`)`

PROPOSED SOLUTION:
$\sigma \to \sigma$
`Type 6` (or `Sort 7`)
`Type 3` (or `Sort 4`)
`Type` $\to$ `Type` (or `Sort 1` $\to$ `Sort 1`)

- 1 point per type

b) A *bag* is a collection of elements that allows multiple (but finitely many) occurrences of its elements. For example, the bag `{2, 7}` is equal to the bag `{7, 2}` but different from `{2, 7, 7}`.

Finite bags can be defined as a quotient over lists. We start with the type `List` α of finite lists and consider only the number of occurrences of elements in the lists, ignoring the order in which elements occur. Following this scheme, `[2, 7, 7]`, `[7, 2, 7]`, and `[7, 7, 2]` would be three equally valid representations of the bag `{2, 7, 7}`.

The `List.count` function returns the number of occurrences of an element in a list. Since it uses equality on elements of type α, it requires α to belong to the `BEq` (Boolean equality) type class. For this reason, the definitions below take `[BEq` α`]` as a type class argument.

```
instance Bag.Setoid (α : Type) [BEq α] : Setoid (List α) :=
{ r     := fun as bs ↦ ∀x, List.count x as = List.count x bs
  iseqv :=
    { refl  := by simp
      symm  := by aesop
      trans := by aesop } }

def Bag (α : Type) [BEq α] : Type :=
  Quotient (Bag.Setoid α)

def Bag.mk {α : Type} [BEq α] : List α → Bag α :=
  Quotient.mk (Bag.Setoid α)
```

Complete the following two definitions. (4 points)

```
def Bag.empty {α : Type} [BEq α] : Bag α :=

def Bag.singleton {α : Type} [BEq α] (a : α) : Bag α :=
```

PROPOSED SOLUTION:

```
Bag.mk []
Bag.mk [a]
```

- 2 points per term

c) Consider the following Lean type classes:

```
universe u

class Mul (α : Type u) : Type u where
  mul : α → α → α

infixl:70 " * " => Mul.mul

class Semigroup (α : Type u)
  extends Mul α : Type u where
  mul_assoc : ∀a b c : α, a * b * c = a * (b * c)
```

The `Semigroup` type class provides an associative binary operator called `mul` abbreviated as `*`.

The concatenation operator ++ on lists is an associative binary operator. Exploit this fact to fill in the gap in the following type class instantiation: (2 points)

```
instance List.Semigroup {α : Type} : Semigroup (List α) :=
  { mul       :=
    mul_assoc := by simp }
```

PROPOSED SOLUTION:
`fun xs ys ↦ xs ++ ys`

2 points for right answer, 0 otherwise