

Solution to the 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 switched off, will be considered a cheating attempt.

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 2**. Questions can be found on **pages 3–18**. There are 7 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	7	Σ
Points	23	16	16	12	16	6	11	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 **rfl**, **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.

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

a) [14 points] Recall the following simplified typing rules for Lean's dependent type theory:

$$\begin{array}{c}
\frac{}{C \vdash c : \sigma} \text{CST} \quad \text{if } c \text{ is globally declared with type } \sigma \\
\\
\frac{}{C \vdash x : \sigma} \text{VAR} \quad \text{if } x : \sigma \text{ is the rightmost occurrence of } x \text{ in } C \\
\\
\frac{C \vdash t : (x : \sigma) \rightarrow \tau[x] \quad C \vdash u : \sigma}{C \vdash t u : \tau[u]} \text{APP}' \\
\\
\frac{C, x : \sigma \vdash t : \tau[x]}{C \vdash (\text{fun } x : \sigma \mapsto t) : (x : \sigma) \rightarrow \tau[x]} \text{FUN}'
\end{array}$$

Let $\text{True} : \text{Prop}$, $h : \mathbb{N} \rightarrow \text{Prop}$, $g : \text{Prop} \rightarrow (\mathbb{N} \rightarrow \text{Prop}) \rightarrow \text{Prop}$, $f : \mathbb{N} \rightarrow \text{List } \mathbb{N} \rightarrow \text{List } \mathbb{N}$ and $ns : \text{List } \mathbb{N}$ be globally declared constants. Specify the types of the following two Lean terms. Give in each case a typing derivation as justification for the type.

(i) $\text{fun } n : \mathbb{N} \mapsto f \, n \, ns$

(9 points)

PROPOSED SOLUTION: The type is $\mathbb{N} \rightarrow \text{List } \mathbb{N}$. The typing derivation is

$$\frac{\frac{\frac{}{n : \mathbb{N} \vdash f : \mathbb{N} \rightarrow \text{List } \mathbb{N} \rightarrow \text{List } \mathbb{N}}{\text{CST}} \quad \frac{}{n : \mathbb{N} \vdash n : \mathbb{N}}{\text{VAR}}}{n : \mathbb{N} \vdash f \, n : \text{List } \mathbb{N} \rightarrow \text{List } \mathbb{N}} \text{APP}' \quad \frac{}{n : \mathbb{N} \vdash ns : \text{List } \mathbb{N}} \text{CST}}{n : \mathbb{N} \vdash f \, n \, ns : \text{List } \mathbb{N}} \text{APP}' \\
\frac{}{\vdash (\text{fun } n : \mathbb{N} \mapsto f \, n \, ns) : \mathbb{N} \rightarrow \text{List } \mathbb{N}} \text{FUN}'$$

- 2 points for type
- 7 points for derivation tree

(ii) $g \, \text{True} \, h$

(5 points)

PROPOSED SOLUTION: The type is Prop . The typing derivation is

$$\frac{\frac{\frac{}{\vdash g : \text{Prop} \rightarrow (\mathbb{N} \rightarrow \text{Prop}) \rightarrow \text{Prop}}{\text{CST}} \quad \frac{}{\vdash \text{True} : \text{Prop}}{\text{CST}}}{\vdash g \, \text{True} : (\mathbb{N} \rightarrow \text{Prop}) \rightarrow \text{Prop}} \text{APP}' \quad \frac{}{\vdash h : \mathbb{N} \rightarrow \text{Prop}} \text{CST}}{\vdash g \, \text{True} \, h : \text{Prop}} \text{APP}'$$

- 2 points for type
- 3 points for derivation tree

b) [9 points] Let α , β , and γ be types in Lean. Give an inhabitant for each of the following types:

(i) $(\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$ (3 points)

PROPOSED SOLUTION: `fun (f :  $\beta \rightarrow \gamma$ ) (a :  $\alpha$ ) (b :  $\beta$ )  $\mapsto$  f b`

- 1 point for `fun f a b`
- 2 points for `f b`

(ii) $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ (3 points)

PROPOSED SOLUTION: `fun (f :  $\alpha \rightarrow \beta$ ) (g :  $\beta \rightarrow \alpha \rightarrow \gamma$ ) (a :  $\alpha$ )  $\mapsto$  g (f a) a`

- 1 point for `fun f g a`
- 2 points for `g (f a) a`

(iii) $(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ (3 points)

PROPOSED SOLUTION: `fun (f :  $\alpha \rightarrow \beta$ ) (g :  $\gamma \rightarrow \beta$ ) (h :  $\beta \rightarrow \gamma$ ) (a :  $\alpha$ )  $\mapsto$  h (f a)`

- 1 point for `fun f g h a`
- 2 points for `h (f a)`

Solution to Question 2 (Functional Programming):**(16 points)**

a) [8 points] Consider the following Lean function definition:

```
def insertBefore (a a₀ : ℕ) : List ℕ → List ℕ
| [] => []
| h :: t => if h = a₀ then a :: h :: insertBefore a a₀ t
            else h :: insertBefore a a₀ t
```

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

(8 points)

```
theorem insertBefore_length (a a₀ : ℕ) (xs : List ℕ) :
  List.length xs ≤ List.length (insertBefore a a₀ xs)
```

PROPOSED SOLUTION: The proof is by structural induction on **xs**.

The base case is

$$[].length \leq (\text{insertBefore } a \ a_0 \ []).length$$

Both sides simplify to 0 and are hence equal.

The induction step is

$$(\text{head} :: \text{tail}).length \leq (\text{insertBefore } a \ a_0 \ (\text{head} :: \text{tail})).length$$

The induction hypothesis is

$$\text{tail}.length \leq (\text{insertBefore } a \ a_0 \ \text{tail}).length$$

If $\text{head} = a_0$, then the induction step simplifies to

$$\text{tail}.length \leq (\text{insertBefore } a \ a_0 \ \text{tail}).length + 1$$

If $\neg \text{head} = a_0$, then the induction step simplifies to

$$\text{tail}.length \leq (\text{insertBefore } a \ a_0 \ \text{tail}).length$$

By the induction hypothesis, both cases hold.

- 1 point for “by 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 (**simp** and **IH**)

- b) [8 points] Define a polymorphic Lean function `mix` that takes two lists and that returns the list that alternates the first `n` elements of the two sequences, where `n` is the length of the shorter sequence. For example, when operating on lists of type `ℕ`, `mix [1, 2] [3, 4]` should evaluate to `[1, 3, 2, 4]`, and so should `mix [1, 2, 5] [3, 4]`. As another example, when operating on lists of type `String`, `mix ["a", "b"] ["c", "d"]` should evaluate to `["a", "c", "b", "d"]`, and so should `mix ["a", "b", "b"] ["c", "d"]`.

PROPOSED SOLUTION:

```
def mix {α : Type} : List α → List α → List α
| h :: t, h' :: t' => h :: h' :: mix t t'
| _ , _ => []
```

- 2 points for “`def mix {α : Type} : List α → List α → List α`” (1 point for polymorphic)
- 2 points for LHS of first equation
- 2 points for RHS of first equation
- 1 point for LHS of second equation
- 1 point for RHS of second equation

Solution to Question 3 (Inductive Predicates):**(16 points)**

- a) [8 points] Consider the following Lean inductive predicate, which determines whether a list is the image of another list under a function f :

```
inductive IsMap {α β : Type} (f : α → β) : List α → List β → Prop
| nil : IsMap f [] []
| cons (xs : List α) (ys : List β) (h : IsMap f xs ys) (a : α):
  IsMap f (a :: xs) (f a :: ys)
```

For example, when α and β are both $\mathbb{N}$, the proposition `IsMap (fun x ↦ x + 2) [5, 3] [7, 5]` should hold, whereas `IsMap (fun x ↦ x + 1) [5, 3] [7, 5]` should not hold.

Prove the following Lean theorem about `IsMap`. Make sure to follow the proof guidelines given on page 2. (8 points)

```
theorem IsMap_length {α β : Type} (xs : List α) (f : α → β) : ∀ys : List β,
  IsMap f xs ys → List.length xs = List.length ys
```

PROPOSED SOLUTION: The proof is by rule induction on `IsMap f xs ys`.

In the `nil` case, the goal is

$$\text{List.length } [] = \text{List.length } []$$

This simplifies to $0=0$, which holds.

In the `cons` case, the goal is

$$\text{List.length } (a :: xs) = \text{List.length } (f a :: ys)$$

The induction hypothesis is

$$\text{List.length } xs = \text{List.length } ys$$

The goal simplifies to

$$(\text{List.length } xs) + 1 = (\text{List.length } ys) + 1$$

which is provable using the induction hypothesis.

- 1 point for “by (rule) induction”
- 1 point for “on `IsMap`”
- 1 point for statement of `nil` case
- 1 point for proof of `nil` case
- 1 point for statement of `cons` case
- 1 point for statement of induction hypothesis
- 2 points for proof of `cons` case (`simp`, `cons`, and IH)

- b) [8 points] Define an inductive predicate **IsBumpy** in Lean that takes a list **xs** of natural numbers and that holds if and only if any two adjacent items in **xs** are not equal. For example, **IsBumpy []**, **IsBumpy [1]**, **IsBumpy [1, 2, 1, 4, 2]** should hold, whereas **IsBumpy [2, 1, 2, 1, 1]** should not hold. (8 points)

PROPOSED SOLUTION:

```
inductive IsBumpy : List ℕ → Prop
| nil : IsBumpy []
| sing (n : ℕ) : IsBumpy [n]
| cons (n n' : ℕ) (l : List ℕ) :
  IsBumpy (n :: l) → n' ≠ n → IsBumpy (n' :: n :: l)
```

- 1 point for “**inductive IsBumpy**”
- 1 point for the **nil** case
- 2 points for the **sing** case
- 1 point for LHS for the **cons** case
- 3 points for RHS for the **cons** case

Solution to Question 4 (Effectful Programming):**(12 points)**

The *accumulator* monad is a monad that accumulates the the input and output values associated with a sequence of operations by returning their sum after the whole sequence of operations. In Lean, it can be defined as follows:

```
def Accum (α : Type) : Type := α × ℕ
def Accum.value {α : Type} : Accum α → α := Prod.fst
def Accum.count {α : Type} : Accum α → ℕ := Prod.snd

def Accum.pure {α : Type} : α → Accum α := fun a ↦ (a, 0)
def Accum.bind {α β : Type} : Accum α → (α → Accum β) → Accum β :=
  fun ma f ↦
    let mb := f (Accum.value ma)
    (Accum.value mb, Accum.count ma + Accum.count mb)
```

- a) [2 points] In addition to **pure** and **bind**, the **Accum** monad should allow its user to associate an input value with an “initial” natural number. Implement the following Lean function accordingly.

```
def Accum.initial {α : Type} (a : α) (n : ℕ) : Accum α
```

PROPOSED SOLUTION:

```
def Accum.initial {α : Type} (a : α)(n : ℕ) : Accum α where
  fst := a
  snd := n
```

- 1 point for **fst := a**
- 1 point for **snd := n**

- b) [6 points] Prove the following law about **Accum** 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. (6 points)

```
lemma Accum.pure_bind {α β : Type} (a : α) (f : α → Accum β) :
  Accum.pure a >>= f = f a
```

PROPOSED SOLUTION: The following sequence of equalities proves the theorem:

```
calc (pure a >>= f)
  = let mb := f (Accum.value (pure a))
    (Accum.value mb, Accum.count (pure a) + Accum.count mb)
    := rfl
  _ = (Accum.value (f (Accum.value (pure a))), Accum.count (pure a) +
Accum.count (f (Accum.value (pure a)))) := rfl
  _ = (Accum.value (f (Accum.value (a, 0))), Accum.count (a, 0) + Accum.count
(f (Accum.value (a, 0)))) := rfl
  _ = (Accum.value (f a), 0 + Accum.count (f a)) := rfl
  _ = (Accum.value (f a), Accum.count (f a)) := by simp
  _ = f a := rfl
```

- 2 points for unfolding of **bind**
- 2 points for unfolding of **pure**
- 2 points for simplification of **match**

- c) [4 points] Commutative monads are monads for which we can reorder actions that do not depend on each other. For the **Accum** monad, the property can be stated as follows:

```
theorem Accum.bind_comm {α β γ δ : Type} (ma : Accum α)
  (f : α → Accum β) (g : α → Accum γ) (h : α → β → γ → Accum δ) :
  ma >>= (fun a ↦ f a >>= fun b ↦ g a >>= fun c ↦ h a b c) =
  ma >>= (fun a ↦ g a >>= fun c ↦ f a >>= fun b ↦ h a b c)
```

Is **Accum** a commutative monad? If yes, provide a justification (as text or in the form of a brief proof sketch). If no, provide a counterexample (i.e., concrete values for **ma**, **f**, **g**, and **h** above and the result of evaluating both sides of the equality).

PROPOSED SOLUTION: Yes, it is a commutative monad. This is because the operation **+** on natural numbers is commutative.

- 1 point for Yes
- 3 points for mentioning that addition is commutative.

Solution to Question 5 (Operational Semantics):**(16 points)**

The ORDER programming language is similar to the familiar WHILE language, with the following difference. The **while-do** statement is replaced by a **order** statement, with the concrete syntax

$$\text{order } b \text{ S T}$$

where b is a boolean value. If b is true, then S is executed first, else T is executed first.

For example, the ORDER program

$$\text{order } b \text{ (j := j + 2) (i := i + 1)}$$

is equivalent to the WHILE program

```
if b then
  (j := j + 2 ; i := i + 1)
else
  (i := i + 1 ; j := j + 2)
```

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

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

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

- a) [8 points] Inspired by the format of the derivation rule for ASSIGN, complete the specification of a small-step semantics for the sequential composition operation **Stmt.seq**. (8 points)

$$\frac{}{(\text{Stmt.assign } x \ a, s) \Rightarrow (\text{skip}, s[x \mapsto s(a)])} \text{ASSIGN}$$

PROPOSED SOLUTION:

$$\frac{}{(\text{skip}; T, s) \Rightarrow (T, s)} \text{SEQ-SKIP} \qquad \frac{(S, s) \Rightarrow (S', s')}{(S; T, s) \Rightarrow (S'; T, s')} \text{SEQ-STEP}$$

- 2 points for SEQ-SKIP
 - 1 point for LHS
 - 1 point for RHS
- 6 points for SEQ-STEP
 - 2 points for assumption
 - 2 points for LHS of the conclusion
 - 2 points for RHS of the conclusion

b) [8 points] Recall that for the inductive predicate:

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

the rule for sequential composition (;) is encoded in a Lean definition as

```
| seq_step (S S' T s s') (hS : SmallStep (S, s) (S', s')) :
  SmallStep (S; T, s) (S'; T, s')
| seq_skip (T s) :
  SmallStep (Stmt.skip; T, s) (T, s)
```

Encode the rules for **order** in a Lean definition of an inductive predicate that encodes the small-step semantics based on subquestion a) above. You are not asked to provide any rules for any other constructor. (8 points)

PROPOSED SOLUTION:

```
| order_true (S T ST' s s') (hS : SmallStep (S; T, s) (ST', s')) :
  SmallStep (Stmt.order true S T, s) (ST',s')
| order_false (S T TS' s s') (hS : SmallStep (T; S, s) (TS', s')) :
  SmallStep (Stmt.order false S T, s) (TS',s')
```

- 4 points for **order_true**
 - 1 point for having the case distinction for the boolean value **true**
 - 1 point for having variables used in the function
 - 1 point for (Stmt.order true S T, s)
 - 1 point for (S ; T, s)
- 4 points for **order_false**
 - 1 point for having the case distinction for the boolean value **false**
 - 1 point for having variables used in the function
 - 1 point for (Stmt.order false S T, s)
 - 1 point for (T ; S, s)

Solution to Question 6 (Logic):**(6 points)**

Consider the two statements below:

$$\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) \leftrightarrow (\mathbf{a} \vee \mathbf{b}) \wedge (\mathbf{a} \vee \mathbf{c})$$

$$\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) \leftrightarrow (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$$

One of the statements is a theorem; the other one is not. Prove the theorem. Make sure to follow the proof guidelines given on page 2.

PROPOSED SOLUTION: The theorem is

$$\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) \leftrightarrow (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$$

Assume $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c})$, then $\mathbf{a}$ holds, and either $\mathbf{b}$ or $\mathbf{c}$ holds. To prove $(\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$, we need either $\mathbf{a} \wedge \mathbf{b}$ or $\mathbf{a} \wedge \mathbf{c}$ to hold. If $\mathbf{b}$ holds, then as $\mathbf{a}$ holds, we have $\mathbf{a} \wedge \mathbf{b}$ holds. If $\mathbf{c}$ holds, then as $\mathbf{a}$ holds, we have $\mathbf{a} \wedge \mathbf{c}$ holds.

Assume $(\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$, then either $\mathbf{a} \wedge \mathbf{b}$ or $\mathbf{a} \wedge \mathbf{c}$. To prove $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c})$, we need both $\mathbf{a}$ and $\mathbf{b} \vee \mathbf{c}$. If $\mathbf{a} \wedge \mathbf{b}$, then we have $\mathbf{a}$ and $\mathbf{b}$. We have $\mathbf{b} \vee \mathbf{c}$ from $\mathbf{b}$. So we have both $\mathbf{a}$ and $\mathbf{b} \vee \mathbf{c}$. If $\mathbf{a} \wedge \mathbf{c}$, then we have $\mathbf{a}$ and $\mathbf{c}$. We have $\mathbf{b} \vee \mathbf{c}$ from $\mathbf{c}$. So we have both $\mathbf{a}$ and $\mathbf{b} \vee \mathbf{c}$.

That proves the theorem.

- 1 point for picking the correct theorem.
- 3 points for LHS to RHS
 - 1 point for splitting the conjunction in $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c})$
 - 1 point for introducing the disjunction in $(\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$
 - 1 point for case analysis on reassociating $\mathbf{b}$ and $\mathbf{c}$ to produce the conjunction $\mathbf{a} \wedge \mathbf{b}$ and $\mathbf{a} \wedge \mathbf{c}$
- 2 points for RHS to LHS
 - 1 point for case analysis on the conjunction in $(\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$
 - 1 point for producing $\mathbf{b} \vee \mathbf{c}$

Solution to Question 7 (Foundations):**(11 points)**

- a) [3 points] Let σ : **Type 5** and τ : **Type 6** be Lean types. Give the type of each of the following Lean terms.

(5 points)

 $\tau \rightarrow \sigma$ **Type 1****fun** (n : $\mathbb{N}$) $\mapsto$ [n, n]**PROPOSED SOLUTION:****Type 6****Type 2** $\mathbb{N} \rightarrow \text{List } \mathbb{N}$

- 1 point per type

- b) [8 points] The byte is a unit of digital information that consists of eight bits. A byte can be represented by a number between 0 and 255. In Lean, such values can be formalized as terms of the type `Mod256`, constructed as follows:

```
instance Mod256.Setoid : Setoid ℕ :=
  { r := fun n1 n2 : ℕ ↦ Nat.mod n1 256 = Nat.mod n2 256
    iseqv := _ }
```

```
def Mod256 : Type := Quotient Mod256.Setoid
```

- (i) [4 points] State and prove the three properties that capture the fact that `r` in the definition of `Mod256` above is an equivalence relation. Make sure to follow the proof guidelines given on page 2.

PROPOSED SOLUTION: The required three properties are reflexivity

$$\forall (x : \mathbb{N}), x.\text{mod } 256 = x.\text{mod } 256$$

symmetry:

$$\forall \{x \ y : \mathbb{N}\}, x.\text{mod } 256 = y.\text{mod } 256 \rightarrow y.\text{mod } 256 = x.\text{mod } 256$$

and transitivity:

$$\forall \{x \ y \ z : \mathbb{N}\}, x.\text{mod } 256 = y.\text{mod } 256 \rightarrow y.\text{mod } 256 = z.\text{mod } 256 \rightarrow x.\text{mod } 256 = z.\text{mod } 256$$

The proof is by the fact that equality is an equivalence relation.

- 1 point for each property of the equivalence relation
- 1 point for pointing out that equality is an equivalence relation

- (ii) [4 points] We define the zero in `Mod256` of the equivalence class of the natural number `0`. In Lean, we set

```
def Mod256.zero : Mod256 := ⟨0⟩
```

Prove the following fact using the definition of `Mod256`

```
theorem Mod256.zero_Eq (m : ℕ) :  
  ⟨m * 256⟩ = Mod256.zero
```

You may use the *soundness* property of quotients. Make sure to follow the proof guidelines given on page 2.

PROPOSED SOLUTION:

```
theorem Mod256.zero_Eq (m : ℕ) :  
  ⟨m * 256⟩ = Mod256.zero :=  
  by  
    rw [Mod256.zero]  
    apply Quotient.sound  
    rw [Mod256.Setoid_Iff]  
    simp
```

- 1 point for expanding the definition of `Mod256.zero`
- 1 point for using soundness
- 2 points for unfolding the definition of being equivalent.