Logical Verification 2021–2022
Vrije Universiteit Amsterdam
Lecturer: dr. J. C. Blanchette

Final Exam
Tuesday 21 December 2021, 08:30–11:15, RAI blok 09
6 questions, 90 points
Answers may be given in English or Dutch

Proof Guidelines

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 **inductive hypotheses** assumed (if any) and the goal to be proved. Unless otherwise specified, minor proof steps corresponding to `refl`, `simp`, or `linarith` need not be justified if you think they are obvious, but you should say which key lemmas they depend on.

You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate, especially for functions and predicates that are specific to an exam question.

Answer:

This version of the exam includes suggested answers, presented in blocks like this one. We present proofs in a textual style, but other styles (e.g., closer to Lean) are also allowed.

In Case of Ambiguities or Errors in an Exam Question

The staff present at the exam has the lecturer's phone number, in case of questions or issues concerning a specific exam question. Nevertheless, we strongly recommend that you work things out yourselves, stating explicitly any ambiguity or error and explaining how you interpret or repair the question. The more explicit you are, the easier it will be for the lecturers to grade the question afterwards.

Question 1. Connectives and quantifiers (6+9 points)

The following two subquestions are about basic mastery of logic. Please provide highly detailed proofs.

1a. Give a detailed proof of the following lemma about universal quantification and disjunction. Make sure to emphasize and clearly label every step corresponding to the introduction or elimination of connective.

```
lemma about_forall_and_or {α : Type} (p q : α → Prop) :
  (∀x, p x) → (∀x, q x) → (∀x, p x ∨ q x)
```

Answer:

Assume `∀x, p x` and `∀x, q x`. Fix `a`. To prove `p a ∨ q a`, by the left introduction rule of ∨, it suffices to prove `p a`. This corresponds to `∀x, p x` instantiated with `a`.

1b. Prove the following one-point rule for existential quantification. In your proof, identify clearly which witness is supplied for the quantifier.

```
lemma exists.one_point_rule {α : Type} {t : α} {p : α → Prop} :
  (∃x : α, x = t ∧ p x) ↔ p t
```

Answer:

By ↔-introduction, it suffices to show
(1) `(∃x : α, x = t ∧ p x) → p t` and
(2) `p t → (∃x : α, x = t ∧ p x)`.

Let's start with (2). Assume `p t`. With the ∃-introduction rule, instantiate `x` with `t`, and then we must prove `t = t ∧ p t`. By ∧-introduction, it suffices to prove `t = t` and `p t`. The first formula is an instance of reflexivity of = and the second formula corresponds to an assumption.

For (1), we assume `∃x : α, x = t ∧ p x` and show `p t`. From the assumption, by ∃-elimination there exists a witness `y` such that `y = t` and `p y`. Using `y = t`, we rewrite `p y` into `p t`, as desired.

Question 2. Lambda-terms (5+5+7 points)

Consider the following inductive type representing untyped λ-terms:

```
inductive lam : Type
| var : string → lam
| abs : string → lam → lam
| app : lam → lam → lam
```

where

- `lam.var x` represents the variable `x`;
- `lam.abs x t` represents the λ-abstraction `λx, t`;
- `lam.app t t'` represents the application `t t'`.

2a. Implement the Lean function

```
def vars : lam → set string
```

that returns the set of all variables that occur freely or bound within a λ-term. For example:

```
vars (lam.var x) = {x}
vars (lam.abs x (lam.var y)) = {x, y}
```

You may assume that the type constructor `set` supports the familiar set operations.

Answer:

```
| (lam.var x)     := {x}
| (lam.abs x t)   := {x} ∪ vars t
| (lam.app t t') := vars t ∪ vars t'
```

2b. Implement the Lean function

```
def free_vars : lam → set string
```

that returns the set of all *free* variables within a λ-term. A variable is free if it occurs outside the scope of any binder ranging over it. For example:

```
free_vars (lam.var x) = {x}
free_vars (lam.abs x (lam.var y)) = {y}
free_vars (lam.abs x (lam.app (lam.var x) (lam.var y))) = {y}
```

Answer:

```
| (lam.var x)     := {x}
| (lam.abs x t)   := free_vars t \ {x}
| (lam.app t t') := free_vars t ∪ free_vars t'
```

2c. Prove that `free_vars` is a subset of `vars`. Note that `A ⊆ B` is defined as `∀a, a ∈ A → a ∈ B`.

```
lemma free_vars_subset_vars (t : lam) :
  free_vars t ⊆ vars t
```

Answer:

The proof is by structural induction on `t`.

Case `var`: The goal is `free_vars (lam.var x) ⊆ vars (lam.var x)`. By definition, both sides simplify to `{x}`. Clearly, `{x} ⊆ {x}`.

Case `abs`: The goal is `free_vars (lam.abs x t) ⊆ vars (lam.abs x t)`. The induction hypothesis is `free_vars t ⊆ vars t`. Simplifying the goal using the definitions above, we get `free_vars t \ {x} ⊆ {x} ∪ vars t`. Clearly, by basic properties of set operations, a sufficient condition for this to hold is if `free_vars t ⊆ vars t`, and this is the induction hypothesis.

Case `app`: The goal is `free_vars (lam.app t t') ⊆ vars (lam.app t t')`. The induction hypotheses are `free_vars t ⊆ vars t` and `free_vars t' ⊆ vars t'`. Simplifying the goal using the definitions above, we get `free_vars t ∪ free_vars t' ⊆ vars t ∪ vars t'`. Clearly, by basic properties of set operations, this follows from the induction hypotheses.

Question 3. A loopy language (8+8 points)

Consider the LOOPY programming language, which comprises three kinds of statements:

- `output s` prints the string `s`;
- `choice S T` nondeterministically executes either `S` or `T`;
- `repeat S` executes `S` a nondeterministic number of times, printing the concatenation (++) of zero or more strings.

In Lean, we can model the language's abstract syntax as follows:

```
inductive stmt : Type
| output : string → stmt
| choice : stmt → stmt → stmt
| repeat : stmt → stmt
```

3a. The big-step semantics for the LOOPY language relates programs `S : stmt` to possible outputs `s : string`.

Complete the following specification of a big-step semantics for the language by giving the missing derivation rules for `choice` and `repeat`.

$$\frac{}{\texttt{output s} \Longrightarrow \texttt{s}}\ \textsc{Output}$$

Answer:

```
S ⟹ s
---------------- ChoiceLeft
choice S T ⟹ s

T ⟹ s
---------------- ChoiceRight
choice S T ⟹ s

--------------- RepeatBase
repeat S ⟹ ""

S ⟹ s    repeat S ⟹ t
------------------------- RepeatStep
repeat S ⟹ s ++ t
```

3b. Specify the same big-step semantics as an inductive predicate by completing the following Lean definition.

```
inductive big_step : stmt → string → Prop
| output {s} : big_step (stmt.output s) s
```

Answer:

```
| choice_left {S T s} : big_step S s → big_step (stmt.choice S T) s
| choice_right {S T s} : big_step T s → big_step (stmt.choice S T) s
| repeat_base {S} : big_step (stmt.repeat S) ""
| repeat_step {S s s'} :
  big_step S s → big_step (stmt.repeat S) s' →
  big_step (stmt.repeat S) (s ++ s')
```

Question 4. The list monad (6+9 points)

The list monad is a monad that stores a list of values of type α. It corresponds to the Lean type constructor `list`.

4a. Complete the Lean definitions of the `pure` and `bind` operations:

```
def list.pure {α : Type} : α → list α

def list.bind {α β : Type} : list α → (α → list β) → list β
```

`pure` should create a singleton list. `bind` should apply its second argument to all the elements of the first argument and concatenate the resulting lists. Examples:

```
list.pure 7 = [7]
list.bind [1, 2, 3] (λx, [x, 10 * x]) = [1, 10, 2, 20, 3, 30]
```

You may assume the following operator and functions are available, among others:

- `++` concatenates two lists;
- `list.map` applies its first argument elementwise to its second argument;
- `list.flatten` transforms a list of list into a flattened list formed by concatenating all the lists together.

Answer:

```
λa, [a]

λas f, list.flatten (list.map f as)
```

4b. Assume `ma >>= f` is syntactic sugar for `list.bind ma f`. Prove the first two monad laws:

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

lemma list.bind_pure {α : Type} (ma : list α) :
  (ma >>= list.pure) = ma
```

Your proofs should be step by step, with at most one rewrite rule or definition expansion per step, so that we can clearly see what happens.

You may assume reasonable lemmas about `list.map` and `list.flatten`. Please state them.

Answer:

For the first property:
`(list.pure a >>= f)`
`= ([a] >>= f)` by definition of `list.pure`

= `list.flatten (list.map f [a])` by definition of `list.bind`
= `list.flatten [f a]` by property of `list.map`
= `f a` by property of `list.flatten`

For the second property: `(ma >>= list.pure)`
= `list.flatten (list.map list.pure ma)` by definition of `list.bind`
= `list.flatten (list.map (λa, [a]) ma)` by definition of `list.pure`
= `list.flatten (list.map (λa, [a]) ma)` by definition of `list.pure`
= `ma` by property `list.flatten (list.map (λa, [a]) as) = as`, provable by induction

Question 5. The loopy language revisited (4+4+6 points)

5a. Implement the following `repeat` function in Lean. It takes a number `n` and a string `s` and returns the string obtained by concatenating `n` copies of `s`.

```
def repeat : ℕ → string → string
```

Answer:

```
| 0       _ := ""
| (n + 1) s := s ++ repeat n s
```

5b. In mathematics, the Kleene star operator takes a string set `A` and returns the set of all the strings that are obtained by concatenating strings from `A` zero or more times. A natural way to model this in Lean is using an inductive predicate. Complete the following definition with the necessary introduction rules so that `kleene_star A s` is true if and only if string `s` is in the Kleene star of set `A`:

```
inductive kleene_star (A : set string) : string → Prop
```

Answer:

```
| empty : kleene_star ""
| step {s t} : s ∈ A → kleene_star t → kleene_star (s ++ t)
```

5c. Use the Kleene star to complete the following definition of the denotational semantics of the LOOPY language from question 3. The denotation of a LOOPY program should be the set of all strings it can output.

```
def denote : stmt → set string
| (stmt.output s) := {s}
```

Recall that the Lean syntax for set comprehensions is `{x | φ x}`, where `φ x` denotes some condition on `x`.

Answer:

```
| (stmt.choice S T) := denote S ∪ denote T
| (stmt.repeat S)   := {s | kleene_star (denote S) s}
```

Question 6. Types and type classes (4+5+4 points)

6a. What are the types of the following expressions?

```
[1, 2, (3 : ℤ)]     list ℕ     list     Sort 1
```

Answer:

`list` $\mathbb{Z}$
`Type`
`Type` $\rightarrow$ `Type` (or `Type u` $\rightarrow$ `Type u`)
`Sort 2`

6b. The type class `monoid` of monoids is defined as follows in Lean:

```
class monoid (α : Type) :=
(mul : α → α → α)
(one : α)
(mul_assoc : ∀a b c : α, mul (mul a b) c = mul a (mul b c))
(one_mul : ∀a : α, mul one a = a)
(mul_one : ∀a : α, mul a one = a)
```

A list can be viewed as a monoid, with the empty list `[]` as `one` and list concatenation `++` as `mul`. Complete the following instantiation of `list` α as a monoid by providing a suitable definition of the five fields of the monoid. For each of the three properties, state the property to prove and very briefly explain why it holds.

```
instance string.monoid {α : Type} : monoid (list α) :=
{ ··· }
```

Answer:

```
{ mul := (++),
  one := [],
  mul_assoc :=
    the property is (a ++ b) ++ c = a ++ (b ++ c),
    i.e. associativity of append on lists, which holds
    (it can be proved by induction),
  mul_one :=
    the property is [] ++ a = a, which clearly holds
    (it is the first case of the definition of ++),
  one_mul :=
    the property is a ++ [] = a, which clearly holds
    (it can be proved by induction) }
```

6c. The type class `group` of groups is defined as follows in Lean:

```
class group (α : Type) :=
(mul : α → α → α)
(one : α)
(mul_assoc : ∀a b c : α, mul (mul a b) c = mul a (mul b c))
(one_mul : ∀a : α, mul one a = a)
(mul_one : ∀a : α, mul a one = a)
(inv : α → α)
(mul_left_inv : ∀a : α, mul (inv a) a = one)
```

Can the type `list` α be instantiated as a group, using the same definition for `mul` and `one` as in question 6b? Briefly explain your answer.

Answer:

No, this cannot be done, because there is no inverse `xs` for `[a]` such that `[a] ++ xs = []`.

The grade for the exam is the total amount of points divided by 10, plus 1.