Logical Verification 2022–2023
Vrije Universiteit Amsterdam
Lecturers: dr. J. C. Blanchette and J. B. Limperg

Resit Exam
14 February 2023, 18:45–21:30, NU-3B19
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.

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+6 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 existential quantification and conjunction. Make sure to emphasize and clearly label every step corresponding to the introduction or elimination of a connective or quantifier.

```
lemma about_exists_and_and {α : Type} (p q : α → Prop) :
  (∃x, p x ∧ q x) → (∃x, p x)
```

Answer:

Assume `hex : ∃x, p x ∧ q x`.
By ∃-elimination, we obtain a witness `x` such that `hand : p x ∧ q x`.
From `hand`, we obtain `hpx : p x` by ∧-left-elimination.
From `hpx`, we obtain `∃x, p x` by ∃-introduction with the witness `x`, as desired. QED

1b. Let `p` be a unary predicate over ℕ. The set of propositions

```
{(∀x, p x), (∃y, ¬ p y)}
```

is inconsistent. Show this by proving the following implication. In your proof, identify clearly which values are supplied for the universal quantifier.

```
lemma inconsistent_predicate {p : ℕ → Prop} :
  (∀x, p x) → (∃y, ¬ p y) → false
```

Answer:

Assume `hall : ∀x, p x`.
Assume `hexn : ∃y, ¬ p y`.
Performing ∃-elimination on `hexn` yields a witness `y` such that `hnpy : ¬ p y`.
From `hall`, we obtain `hpy : p y` by ∀-elimination (by instantiation of `x` with `y`).
From `hnpy` and `hpy`, we derive `false`, as desired. QED

Question 2. Terms of the λ-calculus (6+6+8 points)

Consider the following inductive type representing terms of the untyped λ-calculus:

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

where

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

2a. Implement the Lean function

```
def vars : term → set string
```

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

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

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

Answer:

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

2b. Implement the Lean function

```
def bound_vars : term → set string
```

that returns the set of all *bound* variables within a term. A variable is bound if it is bound by a λ-abstraction. For example:

```
bound_vars (term.var x) = {}
bound_vars (term.abs x (term.var y)) = {x}
bound_vars (term.abs y (term.app (term.var x) (term.var y))) = {y}
```

Answer:

```
| (term.var _)     := {}
| (term.abs x t)   := {x} ∪ bound_vars t
| (term.app t t') := bound_vars t ∪ bound_vars t'
```

2c. Prove that `vars` is never empty:

```
lemma vars_nonempty (t : term) :
  vars t ≠ ∅
```

Answer:

The proof is by structural induction on `t`.

Case `t = term.var x`.
The goal is `vars (term.var x) ≠ ∅`.

By definition of `vars`, the left-hand side of the goal is `vars (term.var x) = {x}`, which is clearly not empty.

Case `t = term.abs x t`.
The goal is `vars (term.abs x t) ≠ ∅`.
The induction hypothesis is `vars t ≠ ∅`.

By definition of `vars`, the left-hand side of the goal is `vars (term.abs x t) = {x} ∪ vars t`, which is clearly not empty.

Case `t = term.app t t'`.
The goal is `vars (term.app t t') ≠ ∅`.
The induction hypotheses are `vars t ≠ ∅` and `vars t' ≠ ∅`.

By definition of `vars`, the left-hand side of the goal is `vars (term.app t t') = vars t ∪ vars t'`.
By the first induction hypothesis, `vars t` is nonempty; hence `vars t ∪ vars t'` is nonempty. QED

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

Consider the STRINGY programming language, which comprises five kinds of statements:

- `skip` does nothing;
- `print s` outputs the string `s`;
- `seq S T` executes `S` then `T`;
- `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
| skip   : stmt
| print  : string → stmt
| seq    : stmt → stmt → stmt
| choice : stmt → stmt → stmt
| repeat : stmt → stmt
```

3a. The small-step semantics for the STRINGY language relates an initial configuration `(S : stmt, s : string)` to a possible next configuration `(S' : stmt, s' : string)`. Notice that strings capture the program's state, in much the same way as `state` in the WHILE language.

Complete the following specification of a small-step semantics for the language by giving the missing derivation rules for the `choice` and `repeat` statements. Notice that `print` appends the new output to any earlier output (using ++).

$$\frac{}{(\texttt{print s'}, \texttt{s}) \Rightarrow (\texttt{skip}, \texttt{s ++ s'})}\ \textsc{Print}$$

$$\frac{(\texttt{S}, \texttt{s}) \Rightarrow (\texttt{S'}, \texttt{s'})}{(\texttt{seq S T}, \texttt{s}) \Rightarrow (\texttt{seq S' T}, \texttt{s'})}\ \textsc{Seq-Step} \qquad \frac{}{(\texttt{seq skip T}, \texttt{s}) \Rightarrow (\texttt{T}, \texttt{s})}\ \textsc{Seq-Skip}$$

Answer:

$$\frac{}{(\texttt{choice S T}, \texttt{s}) \Rightarrow (\texttt{S}, \texttt{s})}\ \textsc{Choice-Left}$$

$$\frac{}{(\texttt{choice S T}, \texttt{s}) \Rightarrow (\texttt{T}, \texttt{s})}\ \textsc{Choice-Right}$$

$$\frac{}{(\texttt{repeat S}, \texttt{s}) \Rightarrow (\texttt{skip}, \texttt{s})}\ \textsc{Repeat-Base}$$

$$\frac{}{(\texttt{repeat S}, \texttt{s}) \Rightarrow (\texttt{seq S (repeat S)}, \texttt{s})}\ \textsc{Repeat-Step}$$

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

```
inductive small_step : (stmt × string) → (stmt × string) → Prop
| print {s s'} :
  small_step (stmt.print s', s) (stmt.skip, s ++ s')
| seq_step {s s' S S' T} :
  small_step (S, s) (S', s') →
  small_step (stmt.seq S T, s) (stmt.seq S' T, s')
| seq_skip {s T} :
  small_step (stmt.seq stmt.skip T, s) (T, s)
...
```

Answer:

```
| choice_left {s S T} :
  small_step (stmt.choice S T, s) (S, s)
| choice_right {s S T} :
  small_step (stmt.choice S T, s) (T, s)
| repeat_base {s S} :
  small_step (stmt.repeat S, s) (stmt.skip, s)
| repeat_step {s S} :
  small_step (stmt.repeat S, s) (stmt.seq S (stmt.repeat S), s)
```

Question 4. A monad for sticky errors (9+6 points)

Error handling can clutter programs. "Sticky errors" are an idiom that can be used to reduce clutter. With sticky errors, a global error flag is initialized to false and is changed to true as soon as an error has occurred. From then on, it remains true, even if some operations succeed—errors are "sticky." At the end of the program, we can check the flag to see if an error has occurred. Using this idiom, we can concentrate the error handling at the end of the program and write the rest of the program optimistically, without worrying about errors. Schematically:

```
do_something();
do_something_else();
if (error_flag)
  print "There was an error";
```

In Lean, we model the state of such a program as a pair `(α, bool)`, where `α` is the result of the program and `bool` indicates whether an error has occurred so far. To increase readability, we introduce aliases for pair constructs:

```
def sticky (α : Type) : Type := (α × bool)
def sticky.value {α : Type} : sticky α → α := prod.fst
def sticky.is_error {α : Type} : sticky α → bool := prod.snd
```

To make it convenient to write programs with sticky errors, we implement the monad operators `pure` and `bind` as follows:

```
def sticky.pure {α : Type} : α → sticky α :=
λa, (a, ff)

def sticky.bind {α β : Type} : sticky α → (α → sticky β) → sticky β :=
λma f,
  let mb := f (sticky.value ma) in
    (sticky.value mb,
     sticky.is_error ma || sticky.is_error mb)
```

For example, the command `#reduce sticky.bind (3, tt) (λn, (7 * n, ff))` prints the `sticky` value `(21, tt)`.

Recall that `tt` and `ff` are the truth values of type `bool`, and `||` denotes "or."

4a. Assume `ma >>= f` is syntactic sugar for `sticky.bind ma f`. Prove the first monad law, shown below. 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 pairs and Booleans. In particular, you may use the lemma stating that `(prod.fst p, prod.snd p) = p` for every pair `p`.

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

Answer:

```
  sticky.pure a >>= f
= (a, ff) >>= f      (by definition of sticky.pure)
= let mb := f a in (sticky.value mb, ff || sticky.is_error mb)
    (by definition of sticky.bind)
= (sticky.value (f a), ff || sticky.is_error (f a))      (by inlining of let)
= (sticky.value (f a), sticky.is_error (f a))      (by basic property of Booleans)
= (prod.fst (f a), prod.snd (f a))      (by definition of sticky.value and sticky.is_error)
= f a      (by lemma (prod.fst p, prod.snd p) = p)
```

4b. Commutative monads are monads for which we can reorder actions that do not depend on each other. For sticky errors, the property can be stated as follows:

```
lemma sticky.bind_comm {α β γ δ : Type} (ma : sticky α) (f : α → sticky β)
    (g : α → sticky γ) (h : α → β → γ → sticky δ) :
  (ma >>= λa, f a >>= λb, g a >>= λc, h a b c) =
  (ma >>= λa, g a >>= λc, f a >>= λb, h a b c)
```

Is `sticky` 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).

Answer:

Yes, `sticky` is a commutative monad. The "value" components of the two computations are the same if we execute `f` or `g` first. It is only the "error" components that are a source of worry, but for both computations, the error flag is set at the end if and only if one of `f`, `g`, or `h` has set it, regardless of the order in which `f` and `g` are executed.

Question 5. The stringy language revisited (6+6 points)

Recall the STRINGY language from Question 3. Based on the small-step semantics we specified in Question 3, we can define a big-step semantics (and the usual syntactic sugar) as follows:

```
infixr ` ⇒ ` := small_step                -- introduces the ⇒ syntax
infixr ` ⇒* ` : 100 := star small_step    -- introduces the ⇒* syntax

def big_step : stmt × string → string → Prop :=
λSs t, Ss ⇒* (stmt.skip, t)

infixr ` ⟹ ` : 110 := big_step            -- introduces the ⟹ syntax
```

Based on the big-step semantics, we can in turn define a Hoare logic (and the usual syntactic sugar):

```
def partial_hoare (P : string → Prop) (S : stmt)
  (Q : string → Prop) : Prop :=
∀s t, P s → (S, s) ⟹ t → Q t

-- introduces the {* P *} S {* Q *} syntax
notation `{* ` P : 1 ` *} ` S : 1 ` {* ` Q : 1 ` *}` := partial_hoare P S Q
```

Notice that `string` takes the place of `state`: Our notion of state consists of a single string, which grows with each `print` statement.

5a. Extend the following specification of the Hoare logic rules for STRINGY by giving the Hoare rules for the `skip` and `seq` statements. (We will return to the `print` statement in Subquestion 5b.)

$$\frac{\{P\}\ S\ \{Q\} \quad \{P\}\ T\ \{Q\}}{\{P\}\ \texttt{choice}\ S\ T\ \{Q\}}\ \textsc{Choice} \qquad \frac{\{I\}\ S\ \{I\}}{\{I\}\ \texttt{repeat}\ S\ \{I\}}\ \textsc{Repeat}$$

Answer:

$$\frac{}{\{Q\}\ \texttt{skip}\ \{Q\}}\ \textsc{Skip} \qquad \frac{\{P\}\ S\ \{Q\} \quad \{Q\}\ T\ \{R\}}{\{P\}\ S;\ T\ \{R\}}\ \textsc{Seq}$$

5b. Which of the following Hoare rule, if any, is correct for the `print` statement? Justify your answer.

$$\frac{}{\{Q\}\ \texttt{print s'}\ \{Q\}}\ \textsc{Print-1}$$

$$\frac{}{\{Q\}\ \texttt{print s'}\ \{\lambda s,\ Q\ (s\ ++\ s')\}}\ \textsc{Print-2}$$

$$\frac{}{\{\lambda s,\ Q\ (s\ ++\ s')\}\ \texttt{print s'}\ \{Q\}}\ \textsc{Print-3}$$

(Note that we mix Lean and informal notations above.)

Answer:

Rule PRINT-3 is correct. If we start in a state `s` that satisfies `Q (s ++ s')` and we extend `s` with `s'`, we end up in a state `s` such that `Q s`. This is analogous to the assignment rule of the WHILE language, with the assignment `s := s ++ s'`.

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

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

```
(3 : ℕ)    ℕ    ℕ → ℕ    list    Type 3
```

Answer:

```
ℕ
Type
Type
Type → Type (or Type u → Type u)
Type 4
```

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

```
@[class] structure 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)
```

Natural numbers can be viewed as a monoid, with `0` as `one` and addition `+` as `mul`. (The names `one` and `mul` are admittedly confusing in this setting.) Complete the following instantiation of $\mathbb{N}$ 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] def nat.monoid : monoid nat :=
{ ... }
```

Answer:

```
mul       := (+),
one       := 0,
mul_assoc := ∀a b c : α, (a + b) + c = a + (b + c),
  -- by associativity of + on natural numbers
one_mul   := ∀a : α, 0 + a = a,
  -- because 0 is neutral element for + on natural numbers
mul_one   := ∀a : α, a + 0 = a
  -- because 0 is neutral element for + on natural numbers
```

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

```
@[class] structure 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 `nat` be instantiated as a group, using the same definition for `mul` and `one` as in Subquestion 6b? Briefly explain your answer.

Answer:

No, because natural numbers other than `0` admit no inverse. For example, there is no natural number `inv 7` such that `(inv 7) + 7 = 0`.

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