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

Final Exam (v3)
Tuesday 20 December 2022, 08:30–11:15, WN-S623 S631 S655
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. Functional programming (5+4+7 points)

1a. Complete the following definition of the function `cycle`:

```
def cycle {α : Type} : ℕ → list α → list α
```

Given a natural number `n` and a list `xs`, `cycle n xs` is a list containing `n` copies of the elements of `xs`. For example:

```
cycle 2 ["I", "love", "exams"] = ["I", "love", "exams", "I", "love", "exams"]
cycle 0 [1, 2] = []
```

You may use the append operator `++`. If you use other functions, give their definitions as well.

Answer:

```
| 0       _  := []
| (n + 1) xs := xs ++ cycle n xs
```

1b. Write the Lean code for a lemma stating that every element of the list `cycle n xs` is also an element of `xs`, for all `n` and `xs`. You do not have to prove the lemma. You can use the binary operator $\in$ of type `Πα, α → list α → Prop` in the lemma statement.

Answer:

```
lemma cycle_mem {α : Type} (n : ℕ) (xs : list α) :
  ∀x : α, x ∈ cycle n xs → x ∈ xs
```

1c. Consider the following `join` function, which concatenates a list of lists:

```
def join {α : Type} : list (list α) → list α
| []          := []
| (xs :: xss) := xs ++ join xss
```

Prove the following lemma about `join` by induction:

```
lemma join_append {α : Type} (xss yss : list (list α)) :
  join (xss ++ yss) = join xss ++ join yss
```

For each case of the induction, clearly indicate the inductive hypotheses assumed and the goal to be proved.

Answer:

The proof is by structural induction on `xss`.

Case `xss = []` (base case).
The goal is `join ([] ++ yss) = join [] ++ join yss`.

For the left-hand side of the goal, we have

```
  join ([] ++ yss)
= join yss          (by definition of ++)
```

For the right-hand side, we have

```
  join [] ++ join yss
= [] ++ join yss    (by definition of join)
= join yss          (by definition of ++)
```

Thus both sides are equal.

Case `xss = xs :: xss'` (inductive step).
The goal is `join ((xs :: xss') ++ yss) = join (xs :: xss') ++ join yss`.
The induction hypothesis is `join (xss' ++ yss) = join xss' ++ join yss`.

For the left-hand side of the goal, we have

```
  join ((xs :: xss') ++ yss)
= join (xs :: (xss' ++ yss))       (by basic property of ++)
= xs ++ join (xss' ++ yss)         (by definition of join)
= xs ++ (join xss' ++ join yss)    (by induction hypothesis)
= xs ++ join xss' ++ join yss      (by associativity of ++)
```

For the right-hand side, we have

```
  join (xs :: xss') ++ join yss
= xs ++ join xss' ++ join yss      (by definition of join)
```

Thus both sides are equal. QED.

Question 2. Logic (8+8 points)

2a. Give a detailed proof of the following lemma. Make sure to emphasize and clearly label every step corresponding to the introduction or elimination of a quantifier or connective.

```
lemma not_exists_forall_not {α : Type} {p : α → Prop} :
  (¬ ∃x, p x) → ∀y, ¬ p y
```

Answer:

Assume `hnex : ¬ ∃x, p x` (i.e., `(∃x, p x) → false`).
Fix `y : α`.
Assume `hpy : p y`.
We must show `false`.
By `hnex`, it suffices to prove `∃x, p x`.
(In other words: We apply `hnex`, giving rise to the subgoal `∃x, p x`.)
At this point, we apply the ∃-introduction rule with the witness `y`.
We must then show `p y`. This follows from `hpy`. QED

2b. Give a detailed proof of the following lemma. Make sure to emphasize and clearly label every step corresponding to the introduction or elimination of a quantifier or connective.

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

Answer:

Assume `hor : (∀x, p x) ∨ (∀x, q x)`.
Fix `x : α`.
We must show `p x ∨ q x`.

Perform an ∨-elimination on `hor`.

Case `hp : ∀x, p x`:
From `hp`, derive `p x` by ∀-elimination (by instantiating `x` with `x`). To prove `p x ∨ q x`, by ∨-left-introduction it suffices to show `p x`, which we have.

Case `hq : ∀x, q x`:
From `hq`, derive `q x` by ∀-elimination (by instantiating `x` with `x`). To prove `p x ∨ q x`, by ∨-right-introduction it suffices to show `q x`, which we have. QED

Question 3. Semantics (4+3+9 points)

We introduce the RNG language, a variant of the WHILE language with nondeterministic variable assignment and a restricted form of `while` loops. It has the following kinds of statements:

- `skip` does nothing;
- `x := [z₁, ..., zₙ]` assigns *one of* the integers z_i, chosen at random, to the variable `x`. If n = 0, the program blocks;
- `S ; T` executes the statement `S` followed by the statement `T`;
- `while_nonzero x do S` executes the statement `S` repeatedly until the variable `x` becomes zero.

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

```
inductive stmt : Type
| skip          : stmt
| assign        : string → list ℤ → stmt
| seq           : stmt → stmt → stmt
| while_nonzero : string → stmt → stmt
```

The infix syntax `S ;; T` abbreviates `stmt.seq S T`.

The big-step semantics of RNG relates a program `S : stmt` and an initial state `s : string → ℤ` with a possible final state `t : string → ℤ`. We write `(S, s) ⟹ t` if the program `S`, when run in the initial state `s`, may terminate in the final state `t`.

3a. Dana Hacker claims that the following derivation rule should be part of the big-step semantics of RNG.

$$\frac{(S,\ s) \Rightarrow t \qquad (T,\ s) \Rightarrow t}{(S\ ;\ T,\ s) \Rightarrow t}\ \textsc{Seq\_Wrong}$$

Explain why this rule is wrong.

Answer:

Both hypotheses start in the same state and end in the same state, executing `S` and `T` in lockstep instead of in sequence.

3b. Despite common sense, Dana wants you to translate her Seq_Wrong rule to Lean. Add an introduction rule corresponding to Seq_Wrong to the following inductive predicate.

```
def state : Type := string → ℤ

inductive big_step : stmt × state → state → Prop
```

Answer:

```
| seq_wrong {S T s t} :
  big_step (S, s) t → big_step (T, s) t → big_step (stmt.seq S T, s) t
```

3c. Complete the following specification of a big-step semantics for RNG by giving the missing derivation rules for `assign`, `seq`, and `while_nonzero`.

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

Answer:

$$\frac{}{\texttt{(x := [z\_1, ..., z\_n], s) => s[x |-> z\_i]}}\ \textsc{Assign}\quad \text{if } \texttt{i} \in \{\texttt{1, ..., n}\}$$

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

$$\frac{}{\texttt{(while\_nonzero x do S, s)} \Rightarrow \texttt{s}}\ \textsc{WhileNonzeroStop}\quad \text{if } \texttt{s(x)} = 0$$

$$\frac{\texttt{(S, s) => t} \qquad \texttt{(while\_nonzero x do S, t) => u}}{\texttt{(while\_nonzero x do S, s)} \Rightarrow \texttt{u}}\ \textsc{WhileNonzeroCont}\quad \text{if } \texttt{s(x)} \neq 0$$

Question 4. List predicates (6+4+8 points)

4a. For a predicate `p` : $\alpha \to$ `Prop` and a list `xs : list` α, we define the predicate `all p xs` that is true if and only if every element of `xs` satisfies `p`. Complete the following definition of `all` as an inductive predicate.

```
inductive all {α : Type} (p : α → Prop) : list α → Prop
```

Answer:

```
| nil          : all []
| cons {x xs} : p x → all xs → all (x :: xs)
```

Alternative:

```
| intro {xs} : (∀x, x ∈ xs → p x) → all xs
```

4b. Similarly, we define the predicate `any p xs` which is true if and only if there exists an element of `xs` that satisfies `p`:

```
inductive any {α : Type} (p : α → Prop) : list α → Prop
| here {x xs}  : p x → any (x :: xs)
| there {x xs} : any xs → any (x :: xs)
```

The same predicate can also be defined using `all`. Give a simple, nonrecursive definition:

```
def anyd {α : Type} (p : α → Prop) (xs : list α) : Prop
```

Answer:

```
:= ¬ all (λx, ¬ p x) xs
```

4c. Yet another way to define `any` is as a recursive function:

```
def anyf {α : Type} (p : α → Prop) : list α → Prop
| []        := false
| (x :: xs) := p x ∨ anyf xs
```

Prove by induction that `any p xs` implies `anyf p xs`. For each case of the induction, clearly indicate the inductive hypotheses assumed and the goal to be proved.

```
lemma any_anyf {α : Type} {p : α → Prop} {xs : list α} :
  any p xs → anyf p xs
```

Answer:

Assume `hany : any p xs`.

The proof is by rule induction on the hypothesis `hany`.

Case `here`:
We have `hp : p x`.
We must show `anyf p (x :: xs)`.

By definition of `anyf`, it suffices to show `p x` ∨ `anyf p xs`.
The left disjunct follows by `hp`.

Case `there`:
We have `hany : any p xs`.
We must show `anyf p (x :: xs)`.
The induction hypothesis is `any p xs` → `anyf p xs`.

By definition of `anyf`, it suffices to show `p x` ∨ `anyf p xs`.
The right disjunct follows by the induction hypothesis and `hany`.

Alternative:

The proof is by structural induction on `xs`.

Case `xs = []` (base case):
The goal is `any p []` → `anyf p []`.

Assume `hany : any p []`.
Perform a case distinction on `hany`.
None of `here` or `there` apply, so there is nothing to prove.

Case `xs = x :: xs'` (induction step):
The goal is `any p (x :: xs)` → `anyf p (x :: xs)`.
The induction hypothesis is `any p xs` → `anyf p xs`.

Assume `hany : any p (x :: xs)`.
Perform a case distinction on `hany`. Two subcases arise:

Subcase `here`:
We have `hp : p x`.
We must show `anyf p (x :: xs)`.
By definition of `anyf`, it suffices to show `p x` ∨ `anyf p xs`.
The left disjunct follows by `hp`.

Subcase `there`:
We have `hany : any p xs`.
We must show `anyf p (x :: xs)`.
By definition of `anyf`, it suffices to show `p x` ∨ `anyf p xs`.
We will prove the right disjunct. By the induction hypothesis, it suffices to show `any p xs`, which corresponds to `hany`. QED

Question 5. Mathematics in Lean (4+5+4+2 points)

5a. What are the types of the following Lean expressions?

```
["What", "is", "my", "type?"]     true     Prop     (λα, set α → list α)
```

Answer:

```
list string
Prop
Type
Type → Type
```
(or `Type u → Type u`)

5b. 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, mul (mul a b) c = mul a (mul b c))
(one_mul   : ∀a, mul one a = a)
(mul_one   : ∀a, mul a one = a)
```

The type of Booleans can be viewed as a monoid, with `ff` as `one` and disjunction `||` as `mul`. Complete the following instance definition accordingly. For the first two fields, give a Lean term. For the other three fields, state the property that needs to be proved to define the field and very briefly explain why it holds.

```
@[instance] def or_monoid : monoid bool :=
{ mul       := ...,
  one       := ...,
  mul_assoc := ...,
  mul_one   := ...,
  one_mul   := ... }
```

Answer:

```
mul       := (||),
one       := ff,
mul_assoc := ∀a b c, (a || b) || c = a || (b || c),
  -- disjunction is associative, as can be seen from a truth table
mul_one   := ∀a, ff || a = a,
  -- falsity is a neutral element for disjunction
one_mul   := ∀a, a || ff = a
  -- falsity is a neutral element for disjunction
```

5c. The relation `same_parity` relates two natural numbers if they are either both even or both odd:

```
inductive same_parity : ℕ → ℕ → Prop
| even {m n} : even m → even n → same_parity m n
| odd {m n}  : odd m → odd n → same_parity m n
```

Prove that `same_parity` is symmetric.

```
lemma same_parity_symm :
  ∀m n : ℕ, same_parity m n → same_parity n m
```

Answer:

Fix `m`, `n : ℕ`.
Assume `hsp : same_parity m n`.
We must show `same_parity n m`.

Perform a case distinction on `hsp`.

Case `even`:
We have `hm : even m` and `hn : even n`.
We apply the introduction rule `same_parity.even` on `hn` and `hm` to show `same_parity n m`.

Case `odd`:
We have `hm : odd m` and `hn : odd n`.
We apply the introduction rule `same_parity.odd` on `hn` and `hm` to show `same_parity n m`. QED

5d. In addition to being symmetric, the relation `same_parity` is also reflexive and transitive, and is therefore an equivalence relation. We use it to form the quotient `parity`:

```
@[instance] def same_parity.rel : setoid ℕ :=
{ r     := same_parity,
  iseqv := ... }

def parity :=
quotient same_parity.rel
```

How many distinct inhabitants does `parity` have? Briefly justify your answer.

Answer:

The type `parity` has two inhabitants, corresponding to the two equivalence classes of the relation `same_parity`. The even numbers form one equivalence class and the odd numbers form the other equivalence class.

Question 6. Monads (4+5 points)

Recall that a monad is lawful if its `pure` and `bind` operations satisfy the three laws given by the `lawful_monad` type class below. We use `ma >>= f` as syntactic sugar for `bind ma f`.

```
@[class] structure lawful_monad (m : Type → Type) [monad m] :=
(pure_bind {α β : Type} (a : α) (f : α → m β) :
   (pure a >>= f) = f a)
(bind_pure {α : Type} (ma : m α) :
   (ma >>= pure) = ma)
(bind_assoc {α β γ : Type} (f : α → m β) (g : β → m γ) (ma : m α) :
   ((ma >>= f) >>= g) = (ma >>= (λa, f a >>= g)))
```

6a. Prove that the following lemma holds for any lawful monad. Your proof should be step-by-step calculational, with at most one rewrite rule per step, so that we can clearly see what happens.

```
lemma pure_bind_bind_bind {m : Type → Type} [monad m] [lawful_monad m]
    {α β γ δ : Type} (a : α) (mb : m β) (f : β → m γ) (g : γ → m δ) :
  pure a >>= (λa, mb >>= (λb, f b >>= g)) = (mb >>= f) >>= g
```

Answer:

  `pure a >>= (λa, mb >>= (λb, f b >>= g))`
= `(λa, mb >>= (λb, f b >>= g)) a` (by `pure_bind`)
= `mb >>= (λb, f b >>= g)` (by β-reduction)
= `(mb >>= f) >>= g` (by `bind_assoc` in reverse)
QED

6b. Does the following statement hold for arbitrary lawful monads? If so, give a proof sketch. If not, give a counterexample and briefly explain why it is a counterexample.

```
lemma reordering {m : Type → Type} [monad m] [lawful_monad m] {α : Type}
    (ma : m α) (f g : α → m α) :
  ((ma >>= f) >>= g) = ((ma >>= g) >>= f)
```

Answer:

As a counterexample, take `m` to be the identity monad, α to be $\mathbb{N}$, `ma` to be `0`, `f` to be `λx, x + 1`, and `g` to be `λx, 2 * x`. Then the left-hand side is equal to `2`, whereas the right-hand side is equal to `1`.

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