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

Resit Exam 1
Tuesday 15 February 2022, 18:45–21:00, NU-3B07
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. Functional programming (3+4+7 points)

Consider the type `btree` of binary trees where each node is either an empty leaf or an inner node with two child trees:

```
inductive btree : Type
| empty : btree
| node  : btree → btree → btree
```

The height of a tree is the largest number of nodes along a path from the root node to a leaf:

```
def height : btree → ℕ
| btree.empty      := 0
| (btree.node l r) := 1 + max (height l) (height r)
```

1a. Complete the definition below of the function `btree.with_height`. Given a natural number `n`, `btree.with_height n` should be a tree of height `n`. You do not need to prove that your definition has the desired property.

```
def btree.with_height : ℕ → btree
```

Answer:

```
| 0 := btree.empty
| (n+1) := btree.node btree.empty (btree.with_height n)
```

1b. Write a Lean definition of an inductive predicate `is_balanced : btree → Prop` that determines if a tree is balanced. A tree is balanced if it is empty or if for each node in the tree, its two subtrees have the same height. You can use the `height` function.

Answer:

```
inductive is_balanced : btree → Prop
| empty : is_balanced btree.empty
| node : ∀l r, is_balanced l → is_balanced r → height l = height r →
    is_balanced (btree.node l r)
```

1c. The `graft` function takes two trees and attaches copies of the second tree to each leaf of the first tree:

```
def graft : btree → btree → btree
| btree.empty      u := u
| (btree.node l r) u := btree.node (graft l u) (graft r u)
```

Give a proof by induction of the lemma `height_graft`, showing that the height of a grafted tree is the sum of the heights of the original trees. For each case, clearly indicate the inductive hypotheses and the goal to be proved. You can refer to the following two lemmas without proving them:

```
#check max_add_add_left  -- ∀a b c : ℕ, max (a + b) (a + c) = a + max b c
#check max_add_add_right -- ∀a b c : ℕ, max (a + b) (c + b) = max a c + b

lemma height_graft :
  ∀t u : btree, height (graft t u) = height t + height u
```

Answer:

The proof is by induction on `t`.

Case `empty`: The goal is `height (graft btree.empty u) = height btree.empty + height u`.

The left-hand side simplifies as follows:
`height (graft btree.empty u)`
`= height u` by definition of `graft`

The right-hand side simplifies as follows:
`height btree.empty + height u`
`= 0 + height u` by definition of `height`
`= height u` by arithmetic

The two sides are equal.

Case `node`. The goal is `height (graft (btree.node l r) u) = height (btree.node l r) + height u`. The induction hypotheses are `∀u, height (graft l u) = height l + height u` and `∀u, height (graft r u) = height r + height u`.

The left-hand side simplifies as follows:
`height (graft (btree.node l r) u)`
`= height (btree.node (graft l u) (graft r u))` by definition of `graft`
`= 1 + max (height (graft l u)) (height (graft r u))` by definition of `height`
`= 1 + max (height l + height u) (height r + height u)` by induction hypotheses
`= 1 + max (height l) (height r) + height u` by `max_add_add_right`

The right-hand side simplifies as follows:
`height (btree.node l r) + height u`
`1 + max (height l) (height r) + height u` by definition of `height`

The two sides are equal.

Question 2. Repeating strings (4+3 points)

2a. Complete the Lean definition of an inductive predicate `is_repeat` on two strings `xs` and `ys`, stating that `ys` is the string `xs` concatenated with itself one or more times:

```
inductive is_repeat : list char → list char → Prop
```

Examples of strings where this relation holds:

```
is_repeat "exams" "exams"
is_repeat "love" "lovelove"
is_repeat "abab" "ababababab"
```

Examples of strings where this relation does not hold:

```
¬ is_repeat "abc" ""
¬ is_repeat "lovelove" "love"
¬ is_repeat "aaa" "aaaa"
¬ is_repeat "abc" "dabcabcd"
```

(For convenience, we identify strings with lists of characters.)

Your definition should consider two cases: `ys` repeats `xs` one time and `ys` repeats `xs` more than one time.

Answer:

```
| once : ∀xs, is_repeat xs xs
| many : ∀xs ys, is_repeat xs ys → is_repeat xs (xs ++ ys)
```

or

```
| once : ∀xs, is_repeat xs xs
| many : ∀xs ys, is_repeat xs ys → is_repeat xs (ys ++ xs)
```

2b. Give a short proof of the proposition `is_repeat "a" "aaa"`:

```
lemma a_aaa : is_repeat "a" "aaa"
```

Answer:

Apply `is_repeat.many` twice, then apply `is_repeat.once`.

Question 3. The IFFY language (6+9+6 points)

The IFFY programming language is similar to the WHILE language, but its `if` statement does not work quite right. It has the following kinds of statements:

- `skip` does nothing;
- `x := a` assigns `a` to the variable `x`;
- `S ; T` executes the statements of `S` followed by the statements of `T`;
- `iffy b then S` does nothing if the Boolean `b` is false. If `b` is true, it nondeterministically chooses between executing the statement `S` or doing nothing.

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

```
inductive stmt : Type
| skip   : stmt
| assign : string → ℤ → stmt
| seq    : stmt → stmt → stmt
| iffy   : bool → stmt → stmt
```

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

3a. The big-step semantics of the IFFY language relates a program `S : stmt` and an input state `s : string` $\rightarrow \mathbb{Z}$ to an output state `t : string` $\rightarrow \mathbb{Z}$. Complete the following big-step semantics by giving the derivation rules for the `iffy` statement:

$$\frac{}{(\texttt{skip},\ \texttt{s}) \Longrightarrow \texttt{s}}\ \textsc{Skip} \qquad \frac{}{(\texttt{assign x a},\ \texttt{s}) \Longrightarrow \texttt{s[x} \mapsto \texttt{s(a)]}}\ \textsc{Asn}$$

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

Answer:

```
       (S, s) ⟹ t
------------------------ If-Exec    if b is true
(iffy b then S, s) ⟹ t

------------------------ If-Skip
(iffy b then S, s) ⟹ s
```

3b. Specify the same big-step semantics in Lean by completing the following definition.

```
inductive big_step : (stmt × (string → ℤ)) → (string → ℤ) → Prop
| skip {s} : big_step (stmt.skip, s) s
```

Answer:

```
| assign {x a s} : big_step (assign x a, s) (λy, if x = y then a else s y)
| seq {S₁ S₂ s t u} : big_step (S₁, s) t → big_step (S₂, t) u →
    big_step (S₁ ;; S₂, s) u
| if_exec {S s t} : big_step (S, s) t → big_step (iffy tt S, s) t
| if_skip {b S s} : big_step (iffy b S, s) s
```

3c. Give a derivation tree in the big-step semantics for an execution of the program `P` defined below, such that the variable `x` gets assigned the value `1`. Clearly indicate the name of each rule.

```
def P : stmt :=
assign "x" 0 ;; iffy true (assign "x" 1)
```

Answer:

```
                                                       ------------------------------------ Assign
                                                       ((assign "x" 1), s[x:=0]) ⟹ s[x:=1]
---------------------------- Assign   -------------------------------------------- If-Exec
(assign "x" 0, s) ⟹ s[x:=0]          (iffy b (assign "x" 1), s[x:=0]) ⟹ s[x:=1]
--------------------------------------------------------------------------------- Seq
       (assign "x" 0 ;; iffy b (assign "x" 1), s) ⟹ s[x:=1]
```

Question 4. Logic (6+6+5 points)

4a. Give a detailed proof of the following lemma. Make sure to emphasize and clearly label every introduction or elimination rule.

```
lemma about_exists_and_or {α : Type} {p q : α → Prop} :
  (∃x, p x ∨ q x) → (∃x, p x) ∨ (∃x, q x) :=
```

Answer:

Assume `h : ∃ x, p x ∨ q x`.
Perform ∃-elimination on `h`, giving a witness `w` and the property `hpq : p w ∨ q w`.
Perform or elimination on `hpq`. This gives two cases.

Case `hp : p w`: Perform ∃-introduction with the witness `w`, giving `∃x, p x`.
Then perform left ∨-introduction to get `(∃x, p x) ∨ (∃x, q x)`.

Case `hq : p w`: Similar to `hp`. Perform ∃-introduction with the witness `w`, giving `∃x, q x`.
Then perform right ∨-introduction to get `(∃x, p x) ∨ (∃x, q x)`.

4b. Let `R : ℤ → ℤ → Prop` be a predicate on two integers that satisfies the following three introduction rules:

```
 R.refl : ∀a, R a a
 R.symm : R ?a ?b → R ?b ?a
R.trans : R ?a ?b → R ?b ?c → R ?a ?c
```

Give a detailed proof of the following theorem. Clearly indicate when you use the rules `R.refl`, `R.symm`, or `R.trans`:

```
theorem euclid : ∀a b c : ℤ, R a b → R a c → R b c
```

Answer:

Fix `a`, `b`, `c`.
Assume `hab : R a b` and `hac : R a c`.
By `R.symm` on `hab`, we get `hba : R b a`.
By `R.trans` on `hba` and `hac`, we get the desired conclusion `R b c`.

4c. One of the following two lemma statements is correct. Indicate which is the correct lemma and give a detailed proof of that statement. Make sure to emphasize and clearly label every introduction or elimination rule.

```
lemma or_of_and {p q : Prop} (h : p ∧ q) : p ∨ q

lemma and_of_or {p q : Prop} (h : p ∨ q) : p ∧ q
```

Answer:

`or_of_and` is correct. By left $\wedge$-elimination, we get `hp : p`. By left $\vee$-introduction, we get the desired conclusion, `p` $\vee$ `q`.

Question 5. Monads (7+5+4+3 points)

5a. Complete the following recursive Lean definition taking a list of functions and a list of arguments. It applies the first function to the first argument, the second function to the second argument, and so on, stopping when either list runs out.

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

For example, `list.pairwise [(λx, x + 1), (λx, x * 2)] [1, 3, 10] = [2, 6]`.

Answer:

```
| []        _         := []
| _         []        := []
| (f :: fs) (x :: xs) := f x :: list.pairwise fs xs
```

5b. Let `m` be a monad. Recall that a monad `m` has two operations:

- `pure {α} : α → m α`
- `bind {α β} : m α → (α → m β) → m β`

Complete the following definition of the operation `ap mf mx` that applies its first boxed argument to the second boxed argument, putting the result in a box:

```
def ap {α β : Type} (mf : m (α → β)) (mx : m α) : m β
```

Answer:

```
do f ← mf,
   x ← mx,
   pure (f x)
```

5c. The operations `pure` and `bind` on `list` are defined as follows:

```
pure x = [x]

bind []        f = []
bind (x :: xs) f = f x ++ bind xs f
```

What are the values returned by the following two calls to `list.length`?

- `list.length (list.pairwise [(λx, x + 1), (λx, x - 1)] [10])`
- `list.length (ap [(λx, x + 1), (λx, x - 1)] [10])`

Answer:

`list.pairwise [(λx, x + 1) (λx, x - 1)] [10] = [10 + 1] = [11]` has length 1.

`ap [(λx, x + 1) (λx, x - 1)] [10] = [(λx, x + 1) (λx, x - 1)] >>= λf, [10] >>= λx, [f x] = ([10] >>= λx, [x + 1]) ++ ([10] >>= λx, [x - 1]) = [11] ++ [9] = [11, 9]` has length 2.

5d. Does applying `ap` to two lists always give the same result as `list.pairwise` on those lists? Briefly explain your answer.

Answer:

No. For exaple, the lengths are different in the previous subquestion.

Question 6. Mathematics in Lean (5+5+2 points)

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

Answer:

Type
ℕ
Type
Type 1
Type

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, 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 `tt : bool` as `one` and the "and" operator `&&` as `mul`. Complete the following instantiation of `bool` 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 bool.monoid : monoid bool :=
{ ... }
```

Answer:

```
{ mul := (&&),
  one := tt,
  mul_assoc :=
    We must show ∀a b c : bool, (a && b) && c = a && (b && c),
    i.e, associativity of conjunction, obvious (by truth table of &&)
  mul_one :=
    We must show ∀a : bool, a && tt = a, obvious (by truth table of &&)
  one_mul :=
    We must show ∀ a : bool, tt && a = a, obvious (by truth table of &&) }
```

6c. The `mathlib` linters reject the following proof. Briefly point out at least one improvement you would make.

```
lemma mul_left_comm {α : Type} [field α]
  (x y z : α) (hx : x ≠ 0) (hy : y ≠ 0) (hz : z ≠ 0) :
  x * (y * z) = y * (x * z) :=
```

```
have hxy : x * y ≠ 0,
  from mul_ne_zero hx hy,
calc  x * (y * z)
    = (x * y) * z : by rw mul_assoc
... = (y * x) * z : by rw mul_comm x y
... = y * (x * z) : by rw mul_assoc
```

Answer:

Possible answers:

- `field` α can become `comm_monoid` or `comm_semigroup`
- the `have hxy` is unnecessary
- the hypothesis `hx`, `hy`, and `hz` can be deleted

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