Logical Verification 2019–2020
Vrije Universiteit Amsterdam
Lecturers: dr. J. C. Blanchette and drs. A. Bentkamp

Final Exam
Tuesday 17 December 2019, 15:15–18:00, WN S-623 S-631 S-655
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. Minor proof steps corresponding to `refl`, `simp`, or `linarith` need not be justified if you think they are obvious (to humans), but you should say which key lemmas they follow from.

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.

In Case of Ambiguities or Errors in an Exam Question

The staff present at the exam has the lecturers' phone numbers, 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. Variable substitution (5+6 points)

Consider this simple type of arithmetic expressions:

```
inductive exp : Type
| Var  : string → exp
| Num  : ℤ → exp
| Plus : exp → exp → exp

export exp (Var Num Plus)
```

1a. Complete the following recursive Lean definition of a function `subst` that performs simultaneous substitution of variables. The application `subst` ρ `e` simultaneously replaces every variable in `e` with a new subexpression: Each occurrence of `Var x` is replaced with ρ `x`.

```
def subst (ρ : string → exp) : exp → exp
```

1b. If we perform simultaneous substitution with `λx, Var x`, we should get the identity function on expressions:

```
lemma subst_Var (e : exp) :
  subst (λx, Var x) e = e
```

Prove this fact by structural induction on `e`. Make sure to follow the proof guidelines given on page 1.

Question 2. Identity monad (4+7 points)

The *identity monad* is a monad that simply stores a value of type α, without any special effects or computational strategy. In other words, it simply provides a box containing a single value of type α. Viewed as a type constructor, the identity monad is the identity function $\lambda\alpha$: `Type`, α. When applying it to a type variable α, we end up with ($\lambda\alpha$: `Type`, α) α, i.e., α itself.

2a. Complete the Lean definitions of the `pure` and `bind` operations:

```
def id.pure {α : Type} : α → α

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

2b. Assume `ma >>= f` is syntactic sugar for `id.bind ma f`. Prove the three monadic laws. Your proofs should be step-by-step calculational, with at most one rewrite rule or definition expansion per step, so that we can clearly see what happens.

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

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

lemma id.bind_assoc {α β γ : Type} (f : α → β) (g : β → γ) (ma : α) :
  ((ma >>= f) >>= g) = (ma >>= (λa, f a >>= g))
```

Question 3. Small-step semantics (6+6+3+3 points)

We introduce WHY, a new programming language that resembles the WHILE language but that includes a few features that make students wonder, "Why??"

The Lean definition of its abstract syntax tree follows:

```
inductive program : Type
| skip {} : program
| havoc   : string → program
| seq     : program → program → program
| choose  : program → program → program
| loop    : program → program

export program (skip havoc seq choose loop)
```

The `skip` and `seq S T` statements have the same syntax and semantics as in the WHILE language. The notation `S ;; T` stands for `seq S T`.

The `havoc x` statement assigns a completely random value to the variable `x`.

The `choose S T` statement randomly executes either `S` or `T`. It behaves like an `if–then–else` statement whose Boolean condition is random.

The `loop S` statement behaves like a `while` loop whose Boolean condition is random. It may exit at any point. For example, `loop S` could have the same end-to-end behavior as `skip` or `S` or `S ;; S` or `S ;; S ;; S` or … .

3a. Complete the following specification of a small-step semantics as derivation rules.

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

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

```
inductive small_step : program × state → program × state → Prop
| seq_step {S T s s' S'} :
  small_step (S, s) (S', s') → small_step (S ;; T, s) (S ;; T, s')
| seq_skip {T s} :
  small_step (skip ;; T, s) (T, s)
```

3c. Is the WHY language deterministic? Briefly explain your answer.

3d. Are WHY programs guaranteed to terminate? Briefly explain your answer.

Question 4. Logical foundations and mathematics (4+2+6+4+2 points)

4a. What are the types of the following Lean constants?

```
true : _     ℕ : _     Prop : _     Type : _
```

4b. Let α : `Type`. The composition `g ∘ f` of two functions `f`, `g` : $\alpha \to \alpha$ is defined as the function λ`x, g (f x)`. What is the neutral element for $\circ$, i.e., the function `h` such that `h ∘ f = f` and `g ∘ h = g` for all `f`, `g` : $\alpha \to \alpha$?

4c. The type class `monoid` 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)
```

Complete the following instantiation of $\mathbb{N} \to \mathbb{N}$ as a `monoid` by providing a suitable definition of `one` and proofs of the `one_mul` and `mul_one` properties:

```
instance monoid_comp : monoid (ℕ → ℕ) :=
{ mul := (λg f, g ∘ f),
  one := _,
  mul_assoc := sorry,
  mul_one := _,
  one_mul := _ }
```

4d. The type class `group` 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 $\mathbb{N} \to \mathbb{N}$ be instantiated as a `group`, using the same definition for `mul` and `one` as in question 4c? Briefly explain your answer.

4e. We used the following definition for Cauchy sequences to construct the real numbers:

```
def is_cau_seq (f : ℕ → ℚ) : Prop :=
∀ε > 0, ∃N, ∀m ≥ N, abs (f N - f m) < ε
```

Which function in this definition needs to be changed to construct the *p*-adic numbers?

Question 5. Palindromes (6+7+6 points)

Palindromes are lists that read the same from left to right and from right to left. For example, `[a, b, b, a]` and `[a, h, a]` are palindromes.

5a. Define an inductive predicate that is `true` if and only if the list passed as argument is a palindrome, by completing the Lean definition below:

```
inductive palindrome {α : Type} : list α → Prop
```

The definition should distinguish three cases:

1. The empty list `[]` is a palindrome.
2. For any element `x` : α, the singleton list `[x]` is a palindrome.
3. For any element `x` : α and any palindrome `[`y_1`, ..., `y_n`]`, the list `[x, `y_1`, ..., `y_n`, x]` is a palindrome.

5b. Let `reverse` be the following operation:

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

Using `reverse`, prove that the reverse of a palindrome is also a palindrome:

```
lemma rev_palindrome {α : Type} (xs : list α) (pal_xs : palindrome xs) :
  palindrome (reverse xs)
```

Make sure to follow the proof guidelines given on page 1. If it helps, you may invoke the following lemma (without having to prove it):

```
lemma reverse_append_sandwich {α : Type} (x : α) (ys : list α) :
  reverse ([x] ++ ys ++ [x]) = [x] ++ reverse ys ++ [x]
```

5c. Prove that the lists `[]`, `[a, a]`, and `[a, b, a]` are palindromes, corresponding to the following lemma statements in Lean:

```
lemma palindrome_nil {α : Type} :
  palindrome ([] : list α)

lemma palindrome_aa {α : Type} (a : α) :
  palindrome [a, a]

lemma palindrome_aba {α : Type} (a b : α) :
  palindrome [a, b, a]
```

Question 6. One-point rules (5+5+3 points)

One-point rules are lemmas that can be used to remove a quantifier from a proposition when the quantified variable can effectively take only one value. For example, `∀x, x = 5 → p x` is equivalent to the much simpler `p 5`.

6a. Prove the one-point rule for ∀. In your proof, identify clearly how the quantifier is instantiated.

```
lemma forall.one_point_rule {α : Type} {t : α} {p : α → Prop} :
  (∀x : α, x = t → p x) ↔ p t
```

6b. Prove the one-point rule for ∃. 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
```

6c. Alyssa P. Hacker proposes the following alternative one-point rule for ∀:

```
axiom forall.one_point_rule' {α : Type} {t : α} {p : α → Prop} :
  (∀x : α, x = t ∧ p x) ↔ p t
```

What is wrong with this rule?

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