Test 2020 final

Test instruction

Welcome to the final exam for the course **Logical Verification** (X_400115).

The only permitted tools are scratch paper, pens, pencils.

Books and other reference materials are **not** permitted.

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 even 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 specified otherwise, 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.

In Case of Ambiguities or Errors in an Exam Question

The lecturers cannot answer questions during the exam. We strongly recommend that you work things out by 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 give you points.

In Case of Late Signup

If you have **not** signed up for this exam, you will not receive a result. Through VUnet you can object to the fact that you can no longer sign up after the expiry of the registration deadline (and the fact that you will not receive a result for this exam). Submit your appeal online within one week after the exam. More information can be found at www.vu.nl/intekenen.

Result text

We hope you enjoyed the course. Please remember to fill out the evaluation if you have not done so already.

Block A: Huffman trees

Question order: Fixed

Consider the following type of weighted binary trees:

```
inductive tree (α : Type)
| leaf : ℕ → α → tree
| inner : ℕ → tree → tree → tree
```

Each constructor corresponds to a kind of node. A `leaf` node stores a numeric weight and a label of some type `α`, and an `inner` node stores a numeric weight, a left subtree, and a right subtree.

Question 1 – 2020 final 1a – 212066.2.1

Define a polymorphic Lean function called `weight` that takes a tree over some type variable `α` and that returns the weight component of the root node of the tree:

```
def weight {α : Type} : tree α → ℕ
```

(Use the "Show block intro" button on the right above to show the introduction to this block of questions, with the definition of `tree`.)

Question 2 – 2020 final 1b – 212075.2.0

Define a polymorphic Lean function called `unite` that takes two trees `l, r : tree α` and that returns a new tree such that (1) its left child is `l`; (2) its right child is `r`; and (3) its weight is the sum of the weights of `l` and `r`.

Question 3 – 2020 final 1c – 212275.2.0

Consider the following `insort` function, which inserts a tree `u` in a list of trees that is sorted by increasing weight and which preserves the sorting. (If the input list is not sorted, the result is not necessarily sorted.)

```
def insort {α : Type} (u : tree α) : list (tree α) → list (tree α)
| [] := [u]
| (t :: ts) :=
  if weight u ≤ weight t then u :: t :: ts else t :: insort ts
```

Prove that `insort`ing a tree into a list cannot yield the empty list:

```
lemma insort_ne_nil {α : Type} (t : tree α) :
  ∀ts : list (tree α), insort t ts ≠ []
```

Block B: Connectives and quantifiers

Question order: Fixed
The following two questions are about basic mastery of logic. Please provide *highly detailed proofs*.

Question 4 – 2020 final 2a – 212271.2.3

Give a detailed proof of the following currying rule. Make sure to emphasize and clearly label every step corresponding to the introduction or elimination of a $\wedge$ or $\leftrightarrow$ connective.

```
lemma currying (p q r : Prop) :
  (p ∧ q → r) ↔ (p → q → r) :=
```

Question 5 – 2020 final 2b – 212281.1.0

Prove the following lemma about quantifiers and implication. Make sure to record and clearly label every step corresponding to the introduction or elimination of a quantifier.

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

Block C: Symmetric and reflexive closures

Question order: Fixed
The following questions are about specifying and reasoning about inductive predicates representing closures of relations.

Question 6 – 2020 final 3a – 212288.1.2

The symmetric closure r^S of a binary relation r over a set A is the smallest relation satisfying the following rules:

(base) for all $a, b \in A$, if $(a, b) \in r$, then $(a, b) \in r^S$;
(symm) for all $a, b \in A$, if $(a, b) \in r$, then $(b, a) \in r^S$.

Complete the following Lean definition of the symmetric closure, in which relations are represented by binary predicates. The definition should follow the structure of the above mathematical definition.

```
inductive sc {α : Type} (r : α → α → Prop) : α → α → Prop
```

Question 7 – 2020 final 3b – 212293.1.1

The reflexive closure $r^?$ of a binary relation r is the smallest reflexive relation that contains r. In Lean, we can define it as follows:

```
inductive rc {α : Type} (r : α → α → Prop) : α → α → Prop
| base (x y : α) : r x y → rc x y
| refl (x : α) : rc x x
```

Prove that if a relation `r` is already reflexive, then `rc r` is the same relation:

```
lemma rc_of_reflexive {α : Type} (r : α → α → Prop)
    (hrefl : ∀x, r x x) (y z : α) :
```

Block D: Logical foundations and mathematics

Question order: Fixed
The following questions concern Lean's logical foundations and its applications to mathematics.

Question 8 – 2020 final 4a – 212323.1.1

What are the types of the following Lean expressions?

```
true
ℤ
list (list ℤ)
list
Type 255
```

Question 9 – 2020 final 4b – 212334.3.0

Recall that any integer *i* can be represented as a pair (*p*, *n*) such that *i* = *p* - *n*, where - denotes integer (nontruncated) subtraction. The pairs (3, 0), (4, 1), and (5, 2) all represent the integer 3, but intuitively (3, 0) seems more appropriate. We will say that a pair (*p*, *n*) is *canonical* if *p* or *n* (or both) is 0:

```
inductive int.is_canonical : ℕ × ℕ → Prop
| nonpos (n : ℕ) : int.is_canonical (0, n)
| nonneg (p : ℕ) : int.is_canonical (p, 0)
```

We can then define the integers as consisting of the canonical pairs of natural numbers:

```
def int : Type :=
{pn : ℕ × ℕ // int.is_canonical pn}
```

Which of the following values are equal? Which are different?

```
def i := subtype.mk (1, 0) (int.is_canonical.nonneg 1)
def j := subtype.mk (0, 0) (int.is_canonical.nonpos 0)
def k := subtype.mk (0, 0) (int.is_canonical.nonneg 0)
def m := subtype.mk (0, 1) (int.is_canonical.nonpos 1)
```

Question 10 – 2020 final 4c – 212339.3.0

Recall the definition of the Lean predicate `even`:

```
inductive even : ℕ → Prop
| zero : even 0
| add_two : ∀k : ℕ, even k → even (k + 2)
```

Also assume the following lemma about `even`:

```
lemma even.add {m n : ℕ} (hm : even m) (hn : even n) :
  even (m + n)
```

Using `even`, we define the type `eveN` of even natural numbers as follows:

```
def eveN : Type :=
{n : ℕ // even n}
```

Define zero and addition of even numbers. Recall that `subtype.mk x h` constructs a subtype value `y` from a base value `x` and a proof `h`, whereas `subtype.val y` and `subtype.property y` retrieve the `x` and `h` components from the subtype value `y`.

```
def eveN.zero : eveN :=
def eveN.add (m n : eveN) : eveN :=
```

Question 11 – 2020 final 4d – 212367.1.0

The following lemma states that any unary operation `f` over `ℕ` has an input `n` at which it returns the minimal element in `f`'s image:

```
lemma exists_minimal_arg (f : ℕ → ℕ) :
  ∃n : ℕ, ∀i : ℕ, f n ≤ f i
```

For example, if `f` is `λn, (n - 100) + (100 - n)`, where – denotes truncated subtraction (i.e., `0 - k = 0`), then the minimum is at `n = 100`.

Use `classical.some : (∃x : ?α, ?p x) → ?α` to define the following Lean function, which returns some input `n` corresponding to the minimal element in `f`'s image.

```
noncomputable def minimal_arg (f : ℕ → ℕ) : ℕ
```

Question 12 – 2020 final 4e – 212368.1.0

Using the representation of *p*-adic numbers as left-infinite streams of digits, compute the following addition in base *p* = 11:

...6666666666 + 1

Block E: Arithmetic expressions

Question order: Fixed

Consider the following type of arithmetic expressions, where `num` represents an integer constant, `var` represents a variable, and `add` represents addition:

```
inductive aexp : Type
| num : ℤ → aexp
| var : string → aexp
| add : aexp → aexp → aexp
```

An environment maps variable names to values:

```
def envir := string → ℤ
```

Question 13 – 2020 final 5a – 212370.2.1

Complete the following Lean definition of an evaluation function for arithmetic expressions.

```
def eval (env : envir) : aexp → ℤ
| (aexp.num i) := i
| (aexp.var x) := env x
```

(Use the "Show block intro" button to show the definition of arithmetic expressions.)

Question 14 – 2020 final 5b – 212374.1.0

Complete the following Lean definition of a big-step-style semantics for arithmetic expressions. The predicate `big_step` () relates an arithmetic expression, an environment, and the value to which it the expression evaluates in the given environment:

```
inductive big_step : aexp × envir → ℤ → Prop
| num {i env} : big_step (aexp.num i, env) i
| add {a₁ a₂ i₁ i₂ env} :
  big_step (a₁, env) i₁ → big_step (a₂, env) i₂ →
  big_step (aexp.add a₁ a₂, env) (i₁ + i₂)


infix `  ` : 110 := big_step
```

On this line, a double arrow ==> is missing

Question 15 – 2020 final 5c – 212385.1.0

Prove that the big-step semantics is sound with respect to the `eval` function:

```
lemma big_step_sound (env : envir) (a : aexp) (i : ℤ) :
  (a, env)  i → eval env a = i
```

On this line, a double arrow ==> is missing

Block F: Metaprogramming

Question order: Fixed
Consider the following tactics `split_hyp` and `split_once`. The main tactic, `split_once`, looks for a hypothesis of the form `h : a ∨ b` in the context, then applies the `cases` tactic to generate two subgoals, one with `h : a` and one with `h : b`.

```
meta def split_hyp (h : expr) : tactic unit :=
do
  T ← tactic.infer_type h,
  match T with
  | `(_ ∨ _) :=
    do
      tactic.cases h,
      pure ()
  | _ := tactic.failed
end

meta def split_once : tactic unit :=
do
  hs ← tactic.local_context,
  list.mfirst split_hyp hs
```

Question 16 – 2020 final 6a – 212398.1.1

For each line of `split_hyp` and `split_once`, write a brief comment explaining what it does.

(Use the "Show block intro" button to show the definition of these two tactics.)

Question 17 – 2020 final 6b – 212404.1.0

Consider the following `case_bash` tactic:

```
meta def case_bash : tactic unit :=
do
  tactic.repeat split_once,
  tactic.all_goals (tactic.try tactic.assumption),
  pure ()
```

Suppose we invoke `case_bash` on the following goal:

```
n m : ℕ,
h : n < m ∨ n = m ∨ n > m
⊢ n < m
```

What subgoal or subgoals does this produce?