Self-Evaluation

Exam from the year 2024

Instructions

- Duration: 3 hours. If you want to try it, set aside some time to work on it without distractions.
- This document has 8 pages and consists of 3 independent exercises as well as a list of Rocq definitions at the end. Read the complete document before starting.
- No machine is allowed. No material is allowed, apart from *one hand-written A4 page* (both sides).
- Language: English or French.
- When asked for code or implementations, write Rocq code, when asked for explanation, write complete English/French sentences. Be concise but precise.
- The Reminders section at the end provides the definition of some standard Rocq constants. The code of any other definition you may use should be explicitly provided in your answer. Implicit types can be omitted or made explicit, at your convenience.
- You may use math notations when writing proof terms: for instance $\forall$ instead of `forall` or λ instead of `fun`.
- You can skip any question and assume previous questions are resolved.
- You can use the `lia` tactic for solving integer arithmetic. Please do not use tactics like `auto`, `intuition` or `firstorder` whose behaviour is less predictable, unless it is explicitly allowed to use them.
- If you write a fixpoint with more than one argument, please be explicit about which one is getting structurally smaller using a {`struct` ...} annotation.

1 Type theory of Rocq

The following questions are independent of each other.

a. Give the proof term for the induction principle for `nat`:
`nat_ind : ∀ (P : nat → Prop), P 0 → (∀ n, P n → P (S n)) → ∀ n, P n`

b. Give the type of the induction principle for the following predicate:

```
Inductive Exists {A : Type} (P : A → Prop) : list A → Prop :=
| Exists_base : ∀ (x : A) (l : list A), P x → Exists P (x :: l)
| Exists_cons : ∀ (x : A) (l : list A), Exists P l → Exists P (x :: l).
```

c. Give the definition of an inductive type `tree A` representing trees whose nodes are labelled in `A` and have a list of children. What would be the type of its *ideal* induction principle? Hint: You may use `Forall` from the appendix.

d. Prove `∀ P Q R, (P ∨ Q) ∧ R → (P ∧ R) ∨ (Q ∧ R)` with a proof term.

e. Prove `∀ n m, le n m → le n (S (S m))` with a proof term.

f. Define a function of type `∀ (A : Type), False → A` and explain why the function is accepted or why it is not possible to define such a function.

g. Can one prove transport for Leibniz equality, i.e. prove the following?
`∀ (A : Type) (u v : A), (∀ (P : A → Prop), P u → P v) → ∀ (Q : A → Type), Q u → Q v`
Note that `Q : A → Type` here! Give a proof script or proof term or explain why it is not possible.

h. Given `A : Type` and `P : A → Prop`, consider the two types `T1 := ∃ x, P x` and `T2 := { x | P x }`.

Define a function of type `T1 → T2`, explain why the function is accepted or why it is not possible.

Define a function of type `T2 → T1`, explain why the function is accepted or why it is not possible.

i. Given `A : Prop` and `B : Prop`, consider the two types `T3 := A ∧ B` and `T4 := A * B`.

Define a function of type `T3 → T4`, explain why the function is accepted or why it is not possible.

Define a function of type `T4 → T3`, explain why the function is accepted or why it is not possible.

j. The induction principle of equality has the following type:
`eq_ind : ∀ (A : Type) (x : A) (P : A → Prop), P x → ∀ (y : A), x = y → P y`
Write a proof term of type `∀ (A : Type) (u v : A), u = v → v = u` by using `eq_ind`. Do not match on equality.

k. What is the shortest proof of `∀ x, false && x = false`?

l. Write a proof term of type `nil ≠ 0 :: nil` either by replacing <todo> in the following code or by using `eq_ind` from above.

```
Definition nil_neq_cons (e : nil = 0 :: nil) : False :=
  match e in _ = l return <todo> with
  | <todo> ⇒ <todo>
  end.
```

m. Write a proof term of type `True ≠ False` either by replacing <todo> in the following code or by using `eq_ind` from above.

```
Definition True_neq_False (e : True = False) : False :=
  match e in _ = X return <todo> with
  | <todo> ⇒ <todo>
  end.
```

2 Using Rocq

The following questions are independent of each other.

a. Finish the following proof. Remember that `lia` does not know anything about `add'`.

```
Fixpoint add' (n m : nat) {struct m} :=
  match m with
  | O => n
  | S m' => add' (S n) m'
  end.

Lemma add'_correct n m : n + m = add' n m.
Proof.
```

b. A monoid is given by a type `A`, some associative operation, and a neutral element for it. Complete the definition of the following class of monoids. You may write ε instead of `mon_neutral` and `x ** y` instead of `mon_op u v` for brevity.

```
Class Monoid A := {
  mon_op : <todo> ;
  mon_assoc : <todo> ;
  mon_neutral : <todo> ;
  mon_left_neutral : <todo> ;
  mon_right_neutral : <todo>
}.
```

Remember that Rocq can automatically figure out instances of a class. You may assume that it works perfectly. Prove the following equality for every monoid.

```
Lemma mon_eq :
  ∀ A (M : Monoid A) (x y z : A),
    (x ** y) ** (ε ** z) = x ** (y ** z).
```

c. Explain whether the following script fails or succeeds and if it does, give the goal together with its hypotheses.

```
Goal ∀ n m, le n m → le (S n) (S m).
Proof.
  intros n m H.
  induction H as [ | m H IH].
  - constructor.
  - constructor.
    (* HERE *)
```

d. Explain whether the following script fails or succeeds and if it does, give the goal together with its hypotheses.

```
Ltac tac :=
  match goal with
  | |- ?A + ?B => left
  | h : ?A |- ?A => apply h
  end.
```

```
Goal ∀ (A B : Type), A → A + B.
Proof.
  intros A B hA.
  tac.
  (* HERE *)
```

e. For each of the following scripts, explain whether it fails or succeeds and if it does, give the goal together with its hypotheses.

```
Goal nat + (bool → bool).
Proof.
  constructor ; intro x.
  (* HERE *)

Goal nat + (bool → bool).
Proof.
  constructor. intro x.
  (* HERE *)
```

f. Recall proposition extensionality and proof irrelevance. Prove that the former implies the latter.

```
Definition PE := ∀ (P Q : Prop), (P ↔ Q) → P = Q.
Definition PI := ∀ (P : Prop) (p q : P), p = q.

Lemma PE_PI : PE → PI.
```

g. Show that accessible points are not inside loops.

```
Lemma Acc_loop (X : Type) (R : X → X → Prop) x :
  Acc R x → R x x → False.
Proof.
  intros Hacc Hx.
```

h. Define the head function for vectors by filling the missing parts:

```
Definition hd {A n} (v : vec A (S n)) : A :=
  match v in vec _ m return <todo> with
  | vnil ⇒ <todo>
  | vcons a v ⇒ <todo>
  end.
```

i. For the following reversal function on lists, give the type of the corresponding functional elimination principle.

```
Equations rev {A} (l : list A) : list A :=
  rev nil := nil ;
  rev (a :: l) := rev l ++ (a :: nil).
```

3 Glivenko's theorem

a. Given the following type of Boolean formulas, what is the type of its induction principle?

```
Inductive form : Type :=
| var : nat → form
| bot : form
| imp : form → form → form.

Notation "s ⇝ t" := (imp s t) (right associativity).
```

b. Can you write a function `size : form → nat` computing the size of a formula? If yes, write it. If no, explain why not.

c. Let the following type of proofs in classical natural deduction be given.

```
Reserved Notation "A ⊢c s" (at level 70).
Inductive ndc A : form → Prop :=
| ndcA s : In s A  →  A ⊢c s
| ndcC s : (s ⇝ bot)::A ⊢c bot  →  A ⊢c s
| ndcII s t : s::A ⊢c t  →  A ⊢c s ⇝ t
| ndcIE s t : A ⊢c s ⇝ t → A ⊢c s  →  A ⊢c t
where "A ⊢c s" := (ndc A s).
```

Can you write a function `size : ∀ A s, A ⊢c s → nat` computing the size of the natural deduction proof proof? If yes, write it. If no, explain why not.

d. Let the following type of proofs in intuitionistic natural deduction be given.

```
Reserved Notation "A ⊢i s" (at level 70).
Inductive nd A : form → Prop :=
| ndA s : In s A  → A ⊢i s
| ndE s : A ⊢i bot  →  A ⊢i s
| ndII s t : s::A ⊢i t → A ⊢i s ⇝ t
| ndIE s t : A ⊢i s ⇝ t  →  A ⊢i s  →  A ⊢i t
where "A ⊢i s" := (nd A s).

Arguments ndIE {A} s t.
```

Also, assume the following weakening lemma.

```
Lemma Weak {A B s} :
  A ⊢i s → incl A B → B ⊢i s.
Admitted.
```

Then prove the following lemma using a proof script.

```
Lemma ndDN A s :
  A ⊢i s → A ⊢i (s ⇝ bot) ⇝ bot.
```

You may use `Weak` without proving it. You may use the tactic `firstorder` to prove goals of the form `In a A` and `incl A B`.

e. Given the following start of a proof script, what are the three remaining goals including hypotheses?

```
Lemma Glivenko A s :
  A ⊢c s → A ⊢i (s ⇝ bot) ⇝ bot.
Proof.
  intros H.
  induction H as [A s H1 |A s _ IH | A s t _ IH | A s t _ IH1 _ IH2].
  - apply ndDN. apply ndA. firstorder.
  - (* HERE *)
  - (* HERE *)
  - (* HERE *)
Qed.
```

f. Prove the first remaining goal, i.e. the second case of the induction.

g. Look at the following proof script and spell out the goal at the marked point.

```
Goal ∀ P Q, (P → ¬¬Q) → ¬¬(P → Q).
Proof.
  intros P Q PinnQ nPiQ.
  apply nPiQ.
  intros p. exfalso. apply PinnQ.
  - apply p.
  - intros q.
    (* HERE *)
    apply nPiQ. intros _. exact q.
Qed.
```

h. Prove the second remaining goal of `Glivenko`, i.e. the third case of the induction. The given Rocq script from **g.** above will help you. You may use `Weak` without proving it. You may use the tactic `firstorder` to prove goals of the form `In a A` or `incl A B`.

i. Consider the following proof script of the last goal of `Glivenko`.

```
- apply ndII. apply (ndIE (s ⇝ bot)).
  + apply (Weakm IH2). firstorder.
  + apply ndII.
    apply (ndIE ((s ⇝ t) ⇝ bot)).
    * apply (Weakm IH1). firstorder.
    * apply ndII.
      apply (ndIE t). 1:{ eapply ndA. firstorder. }
      apply (ndIE s). 1:{ eapply ndA. firstorder. }
      eapply ndA. firstorder.
```

Take inspiration from it to prove the following goal.

```
Goal ∀ P Q, ¬¬(P → Q) → ¬¬P → ¬¬Q.
```

Reminders

You can use the following Rocq definitions together with their induction principles. All other types or functions you need have to be defined.

```
Inductive True : Prop := I.

Inductive False : Prop :=.

Inductive eq (A : Type) (x : A) : A → Prop :=
| eq_refl : eq A x x.

Notation "x = y" := (eq _ x y).
Notation "x ≠ y" := (eq _ x y → False).

Notation "¬ P" := (P → False).

Inductive bool :=
| true | false.

Definition andb (b c : bool) : bool :=
  if b then c else false.
Notation "b && c" := (andb b c).

Inductive nat :=
| O | S (n : nat).

Fixpoint add (n m : nat) {struct n} :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (add n' m)
  end.
Notation "n + m" := (add n m).

Inductive list (A : Type) : Type :=
| nil
| cons (a : A) (l : list A).

Arguments nil {A}.
Arguments cons {A} a l.

Notation "a :: l" := (cons a l).

Fixpoint In {A} (a:A) (l:list A) : Prop :=
    match l with
      | nil ⇒ False
      | b :: m ⇒ b = a ∨ In a m
    end.

Definition incl {A} (l l' : list A) :=
  ∀ x, In x l → In x l'.

Fixpoint app {A : Type} (l1 l2 : list A) {struct l1} : list A :=
  match l1 with
  | nil ⇒ l2
  | a :: l ⇒ a :: app l l2
  end.
Notation "l1 ++ l2" := (app l1 l2) (right associativity).
```

```
Inductive or (A B : Prop) : Prop :=
| or_introl (a : A) : or A B
| or_intror (b : B) : or A B.

Notation "A ∨ B" := (or A B).

Inductive and (A B : Prop) : Prop :=
| conj (a : A) (b : B).

Notation "A ∧ B" := (and A B).

Inductive prod (A B : Type) : Type :=
| pair (a : A) (b : B).

Notation "A * B" := (prod A B).

Inductive ex (A : Type) (P : A → Prop) : Prop :=
| ex_intro (a : A) (p : P a).

Notation "∃ x, P" := (ex A (fun x ⇒ P)).

Inductive sig (A : Type) (P : A → Prop) : Type :=
| exist (a : A) (p : P a).

Notation "{ x | P }" := (sig A (fun x ⇒ P)).

Inductive Forall {A : Type} (P : A → Prop) : list A → Prop :=
| Forall_nil : Forall P nil
| Forall_cons : ∀ (x : A) (l : list A), P x → Forall P l → Forall P (x :: l).

Inductive le (n : nat) : nat → Prop :=
| le_n : le n n
| le_S m : le n m → le n (S m).

Notation "n ≤ m" := (le n m).

Definition iff (A B : Prop) :=
  (A → B) ∧ (B → A).

Notation "A ↔ B" := (iff A B).

Inductive Acc {A} (R : A → A → Prop) : A → Prop :=
| Acc_intro : (∀ y, R y x → Acc R y) → Acc R x.

Inductive vec (A : Type) : nat → Type :=
| vnil : vec A 0
| vcons (a : A) (n : nat) (v : vec A n) : vec A (S n).

Arguments vnil {A}.
Arguments vcons {A} a {n}.

Inductive sum (A B : Type) : Type :=
| inl : A → A + B
| inr : B → A + B
where "A + B" := (sum A B).
```