Project – Natural Deduction, Heyting Semantics, and Proof by Reflection

v1.1, October 18th

Please hand in a zip file containing the three exercises called `ex1.v`, `ex2.v` and `ex3.v` as well as a PDF report. Our emails: `yannick.forster@inria.fr` and `theo.winterhalter@inria.fr`.

Ask for help on the Zulip. Ensure that your project builds with Coq `8.16.1`. You are allowed to use packages such as Equations or MetaCoq as included in the release of the Coq platform version `2022.09.1` at the *extended level*. To help us read your project, please identify to which answer you reply by using comments with questions numbers such as `(* 1.2.b *)`.

Write the report in PDF format with explanation of design choices and difficulties you encountered. At the start of your report, please include an assessment of your previous experience with Coq or other proof assistants. You can write the report in English or French.

One piece of advice: Take a step back whenever you are stuck. Doing Coq proofs can sometimes feel like a video game. If that happens, maybe you need to take a break to reflect on how you want to prove the thing. It might also help to do it on paper in those cases.

Deadline: 20 November 2023 at 21:00.

1 Natural Deduction

In this exercise, we will define natural deduction systems for both intuitionistic and classical propositional logic. We show that provability of ground formulas is decidable, that the classical and the intuitionistic system are equiconsistent, and that the intuitionistic system does not prove double negation elimination, which entails that it is different from the classical system and that both are consistent.

Mathematically, the intuitionistic natural deduction system we consider has 4 rules (assumption, explosion, implication introduction, implication elimination):

$$\frac{s \in A}{A \vdash s} \qquad \frac{A \vdash \bot}{A \vdash s} \qquad \frac{s, A \vdash t}{A \vdash s \to t} \qquad \frac{A \vdash s \qquad A \vdash s \to t}{A \vdash t}$$

1.1 Intuitionistic

Start a file `ex1.v` with the following definitions and Notations.

```
Require Import List.
Import ListNotations.

Inductive form : Type :=
| var (x : nat) | bot | imp (s t : form).

Print In.
Print incl.

Notation "s ~> t" := (imp s t) (at level 51, right associativity).
Notation neg s := (imp s bot).
Reserved Notation "A ⊢ s" (at level 70).
```

a. Define an inductive predicate `nd : list form → form → Prop` capturing the rules from above. Declare the notation `A ⊢ s` for `nd A s`.

b. Construct natural deduction proofs of the following statements:

1. `A ⊢ s ~> s`
2. `s :: A ⊢ neg (neg s)`
3. `[neg (neg bot)] ⊢ bot`

c. Prove weakening:

```
Fact Weak A B s :
  A ⊢ s → incl A B → B ⊢s.
```

d. Define a predicate `ground : form → Prop` ensuring that no variables occur in a formula. Prove that ground formulas are decidable:

```
Fact ground_decidable s :
  ground s → ([] ⊢ s) + ([] ⊢ neg s).
```

1.2 Classical

a. Classical natural deduction can be defined almost the same way by replacing the explosion rule by the rule below. Define a predicate `ndc : list form → form → Prop` with notation `A ⊢c s`.

$$\frac{\neg s :: A \vdash_c \bot}{A \vdash_c s}$$

b. Prove `A ⊢c (neg (neg s)) ~> s`.

c. Prove

```
Lemma Weakc A B s :
  A ⊢c s → incl A B → B ⊢c s.
```

d. Prove that intuitionistically provable formulas are classically provable:

```
Lemma Implication A s :
  A ⊢ s → A ⊢c s.
```

e. Define the Friedman translation `trans : form → form → form` such that `trans t s` replaces every occurrence of `bot` in `s` by `t` and `var x` by `(var x ~> t) ~> t`.

f. Prove

```
Lemma DNE_Friedman A s t :
  A ⊢ ((trans t s ~> t) ~> t) ~> (trans t s).
```

g. Prove

```
Lemma Friedman A s t :
  A ⊢c s → map (trans t) A ⊢ trans t s.
```

h. Deduce that intuitionistic and classical natural deduction derive the same ground formulas:

```
Lemma ground_truths s :
  ground s → ([] ⊢ s ↔ [] ⊢c s).
```

i. Deduce that intuitionistic natural deduction is consistent if and only if classical natural deduction is consistent.

1.3 Consistency

We are going to prove that intuitionistic natural deduction is consistent by showing that it cannot derive double negation elimination. To do so, we are going to define a semantics for propositional logic with three truth values.

```
Inductive tval := ff | nn | tt.

Definition leq a b : bool :=
  match a, b with
  | ff , _ ⇒ true
  | nn, nn ⇒ true
  | nn, tt ⇒ true
  | tt, tt ⇒ true
  | _, _ ⇒ false
  end.

Notation ”a <= b” := (leq a b).

Definition impl a b : tval :=
  if a <= b then tt else b.

Fixpoint eva alpha (s : form) : tval :=
  match s with
  | var x ⇒ alpha x
  | bot ⇒ ff
  | imp s1 s2 ⇒ impl (eva alpha s1) (eva alpha s2)
  end.

Fixpoint evac alpha (A : list form) : tval :=
  match A with
  | nil ⇒ tt
  | s:: A’ ⇒ if eva alpha s <= evac alpha A’ then eva alpha s else evac alpha A’
  end.

Notation leb alpha A s := (evac alpha A <= eva alpha s = true).
```

a. Prove soundness of this semantics.

```
Theorem nd_sound alpha A s :
  A ⊢ s → leb alpha A s.
```

b. Prove that intuitionistic natural deduction does not derive double negation elimination, i.e.

```
Corollary nd_DN x :
  ~[] ⊢ (neg (neg (var x))) ~> (var x).
```

c. Prove that intuitionistic natural deduction is consistent, i.e.

```
Corollary nd_consistent :
  ~[] ⊢ bot.
```

d. Prove that classical natural deduction is consistent, i.e.

```
Corollary ndc_consistent :
  ~[] ⊢c bot.
```

2 Heyting Semantics [Doable from lesson 5]

This exercise is an extension of the previous exercise. You will extend formulas with conjunction and generalise the three-point semantics used in the last exercise to a general semantics for which natural deduction is sound and complete.

Start with a fresh Coq file called `ex2.v`. Copy-pasting old proofs is allowed. We recommend the following boilerplate to start the file:

```
Set Implicit Arguments.
Unset Strict Implicit.
Require Import List.
Import ListNotations.
```

a. Extend the type of formulas with conjunction, extend the `nd` predicate with an elimination and introduction rule, and re-prove weakening.

b. A Heyting algebra consists of:

- A type H,
- a reflexive, transitive relation $\leq$ on H.
- a constant $\bot : H$,
- an operation $\sqcap : H \to H \to H$, called *meet*, and
- an operation $\Rightarrow: H \to H \to H$, called *implication*,

such that

1. $\bot \leq u$
2. $u \leq s \land u \leq t \leftrightarrow u \leq s \sqcap t$
3. $(s \sqcap t) \leq u \leftrightarrow s \leq (t \Rightarrow u)$.

Define the type of Heyting algebras in Coq using a `Record HeytingAlgebra`.

c. Define an evaluation function

```
eval : ∀ (HA : HeytingAlgebra), (nat → H HA) → form → H HA
```

evaluating a formula in a given Heyting algebra.

d. Define a function

```
Meet_list : ∀ HA, (nat → H HA) → list form → H HA
```

such that e.g. `Meet_list [x1, x2, x3] = x1 ⊓ x2 ⊓ x3`.

e. We define `Definition hent HA interp A s := (@Meet_list HA interp A) <= (@eval HA interp s)`. Prove

```
Lemma nd_soundHA A s : nd A s → hent A s.
```

f. Define a function `revert : list form → form → form` and prove

```
Lemma revert_correct A s :
  A ⊢ s ↔ [] ⊢ revert A s.
```

g. Show that one can build a Heyting algebra out of formulas, with the relation defined as `fun s t ⇒ [] ⊢ s ~> t`.

h. Prove that evaluation in this Heyting algebra is the identity.

i. Prove completeness of Heyting semantics to deduce the following:

```
Theorem HA_iff_nd A s:
  (∀ (HA : HeytingAlgebra) (V : nat → H HA), hent V A s) ↔ nd A s.
```

Possible exercises for bonus points

Choose one or several of the following.

- Use advanced techniques you have learned in the course (Equations, type classes, setoid rewriting, hint-based automation, Ltac automation, etc.) to simplify your development.
- Extend formulas e.g. by disjunction (relatively easy) or quantifiers (hard, ask us for advice first if you want to go that route), then re-prove all theorems. Please submit a separate file for this.
- Prove other properties or theorems you can come up with. For instance, you could prove that disjunction is not expressible using only the other connectives. Alternatively, you could also prove that both intuitionistic natural deduction (e.g. via tableaux methods) and classical natural deduction (e.g. via a sound and complete boolean semantics) are decidable. Again, please submit a separate file.

3 Reflection

The goal is to write an automatic tactic for proving that two booleans formulas are equal, by converting them to polynomial formulas over $\mathbb{Z}$ and checking that both polynomials are equal using the `lia` tactic. Write the solutions to this exercise to a new file called `ex3.v` which can start by requiring $\mathbb{Z}$ arithmetic:

```
Require Import ZArith.
```

This gives you the type `Z` of integers.

a. Extend the formula type from the last exercise to include disjunction.

b. Define a function that transforms a formula into a number of type $\mathbb{Z}$:

- $\overline{\bot} = 0$
- $\overline{a \wedge b} = \overline{a} \times \overline{b}$,
- $\overline{a \vee b} = \overline{a} + \overline{b} - \overline{a} \times \overline{b}$,
- $\overline{a \rightarrow b} = \overline{a} \times \overline{b} - \overline{a} + 1$

It will have to take a valuation from the variables, ie. a map from `nat` to `Z`.

Prove that evaluating a formula f into booleans gives the same result as evaluating its transformation $\overline{f}$ into $\mathbb{Z}$.

c. Deduce a process for automatically proving boolean tautologies in Coq using `lia`. Test it on various boolean equalities, e.g. $\neg a \vee \neg b \vee (c \wedge \top) = \neg(a \wedge b \wedge \neg c)$.

Below you can find examples how to implement reification in Ltac or MetaCoq. The reification only treats the negation case, you are expected to fill in the other cases. [Note: Some of it will be explained in lesson 7 so don't worry if you don't manage to do it yet.]

d. (Bonus question) How complete is the process? (That is, are there actual boolean equalities that cannot be proved by your tactic?) If so, how can this shortcoming be avoided?

e. (Bonus question) Extend your tactic to also handle hypotheses in the context.

Ltac template

```
Ltac list_add a l :=
  let rec aux a l n :=
    lazymatch l with
    | [] ⇒ constr:((n, cons a l))
    | a :: _ ⇒ constr :((n, l))
    | ?x :: ?l ⇒
        match aux a l (S n) with
        | (?n , ?l ) ⇒ constr:((n, cons x l))
        end
    end in
  aux a l O.

Ltac read_term f l :=
   lazymatch f with
  | negb ?x ⇒
    match read_term x l with
    | (?x', ?l') ⇒ constr:((imp x' bot, l'))
    end
 (* fill in other cases here *)
 | _ ⇒
     match list_add f l with
     | (?n, ?l') ⇒ constr:((var n, l'))
     end
  end.

Ltac reify f :=
  read_term f (@nil bool).
```

MetaCoq template

```
Fixpoint index {A} (d : A → A → bool) a l :=
  match l with
  | [] ⇒ None
  | x :: l ⇒
      if d x a then Some 0 else
      match index d a l with
      | Some n ⇒ Some (S n)
      | None ⇒ None
      end
  end.

Definition list_add {A} (d : A → A → bool) a l :=
  match index d a l with
  | Some n ⇒ (n, l)
  | None ⇒ (length l, l ++ [a])
  end.

Fixpoint read_term' f l : TemplateMonad (form * list term) :=
  let catchall :=
    fun _ : unit ⇒
```

```
    let '(n, l') := list_add (@eq_term config.default_checker_flags init_graph) f l in
    ret (var n, l') in
  match f with
    | tApp (tConst cst []) [x] ⇒
      if eqb cst (MPfile ["Datatypes"; "Init"; "Coq"], "negb")
      then
        mlet (x', l') ← read_term' x l ;; ret (imp x' bot, l')
      else
        catchall tt
    (* fill in other cases here *)
    | _ ⇒
      catchall tt
  end.

Ltac reify f :=
  constr:(ltac:(
    run_template_program (mlet t ← tmQuote f ;;
                          mlet (x, lx) ← read_term' t (@nil term) ;;
                          mlet lx' ← monad_map (tmUnquoteTyped bool) lx ;;
                          ret (x, lx'))
      (fun x ⇒ exact x))).
```

Changelog

Changes from `v1.0` (Oct 9th) to `v1.1`

- Changed the suggested notation for `nd` to be unicode ⊢ rather than |−, to avoid clashes with the `match goal with [ H : _ |- _ ] ⇒ t end.` Ltac construct. Thanks to Neven Vilani for spotting this issue.
- Changed the suggested definition of hent to be

  ```
  Definition hent HA interp A s := (@Meet_list HA interp A) <= (@eval HA interp s).
  ```

  Before, the arguments `HA` and `interp` were missing. Using `@` avoid problems with implicit arguments.

  Thanks to Félix Ridoux and Salwa Tabet Gonzaled for spotting this issue.