Master Parisien de Recherche en Informatique (MPRI) 2024–2025
Proof Assistants (2-7-2)

# Consistency of classical propositional natural deduction

`v1.6, 8 November`

Please hand in a zip file containing the three exercises called `ex1.v`, `ex2.v` and `ex3.v`. The subject of your email should be "[MPRI 2-7-2] Project". Our emails: `yannick.forster@inria.fr` and `theo.winterhalter@inria.fr`.

**Asking for help.** It is explicitly allowed to ask for help on the course's Discord when you get stuck regarding Coq-specific problems. Do not ask for solutions, but you can ask about concrete issues you have. Furthermore, it is allowed to discuss problems with your fellow students outside of the Discord, but:

- These discussions are only allowed to happen verbally, *i.e.* never in written form. In particular, this excludes discussion via email, text message, any Discord that is not the course's Discord, etc. Really, make sure that you do *not* share code.
- Please start your `ex1.v` file with a comment explaining who you discussed what with. This includes both questions you ask fellow students *and* help you give to fellow students.
- You are not allowed to use tools such as `ChatGPT` or similar tools.

**Building projects.** Ensure that your project builds with Coq `8.18.0`. You are allowed to use packages such as Equations or MetaCoq as included in the release of the Coq platform version `2023.11.0` at the *extended level.* To help us read your project, please identify to which answer you reply to by using comments with questions numbers such as `(* 1.2.b *)`.

**Comments.** Please add comments to your Coq file explaining design choices and difficulties you encountered.

**Evaluation.** By solving all mandatory exercises, you can obtain 16 out of 20 points. To get more than this, look at the mini projects to choose from in section 3.

**Previous experience.** At the start of file `ex1.v`, please include an assessment of your previous experience with Coq or other proof assistants.

**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.

**Changes.** We will publish new versions of this PDF in case it becomes necessary, *i.e.* fixing typos or mistakes. We will update the version number and attach a changelog. You will of course be notified by email and on the Discord.

- `v1.6`. Fix typo in 2.4: In the paragraph starting with "For (2)", there were mentions of some $e_2$ which was actually $e_1$.

- `v1.5`. Fix typo in 2.1.d: Instead of "…then $A \vdash_H \ s \to t$." it has to be "…then $A \vdash_H s \to t$."
- `v1.4`. Fix typos in 2.1.b, 2.1.e where `nd` was used instead of `ndm`. Add **`End`** `ARS`. before 2.2.e. Changed the **`Import`** in 2.3 to specify that `ex1` needs to be imported.
- `v1.3`. Fix typos in the notation of 2.3, in the rules of 2.3.a, in the notation of 2.3.c and the statement of 2.3.d.
- `v1.2`. Fix typo in type of `typing` in 2.3.a.
- `v1.1`. Fix typo in the `"A ⊢c s"` notation of 1.1.
- `v1.0`. Initial version.

**Deadline:** 14 November 2024 at 18:00.

## 1 Natural Deduction

In this exercise, we will define a natural deduction systems for both classical and minimal propositional logic.

### 1.1 Classical

Mathematically, the classical natural deduction system we consider has 4 rules (assumption, implication introduction, implication elimination, proof by contradiction):

$$\frac{s \in A}{A \vdash_c s} \qquad \frac{s, A \vdash_c t}{A \vdash_c s \rightarrow t} \qquad \frac{A \vdash_c s \rightarrow t \qquad A \vdash_c s}{A \vdash_c t} \qquad \frac{\neg s :: A \vdash_c \bot}{A \vdash_c s}$$

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 ⊢c s" (at level 70).
```

**a.** Define an inductive predicate `ndc : list form -> form -> Prop` capturing the rules from above. Declare the notation `A ⊢c s` for `ndc 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`. Can you do it without using proof by contradiction?
4. `A ⊢c (neg (neg s)) ~> s`

**c.** Prove weakening:

```
Fact Weakc A B s :
  A ⊢c s -> incl A B -> B ⊢c s.
```

**d.** Define a predicate `ground : form -> Prop` ensuring that no variables occur in a formula.

## 1.2 Minimal

**a.** Minimal natural deduction can be defined by removing the rule for proofs by contradiction from natural deduction. Note that in particular, there is not even an explosion rule. Define it as a predicate `ndm : list form -> form -> Prop` with notation `A ⊢m s`.

**b.** Prove

```
Lemma Weakm A B s :
  A ⊢m s -> incl A B -> B ⊢m s.
```

**c.** Prove that minimally provable formulas are classically provable:

```
Lemma Implication A s :
  A ⊢m s -> A ⊢c s.
```

**d.** 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`.

**e.** Prove

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

**f.** Prove

```
Lemma Friedman A s t :
  A ⊢c s -> map (trans t) A ⊢m trans t s.
```

**g.** Deduce that minimal and classical natural deduction derive the same ground formulas:

```
Lemma ground_truths s :
  ground s -> ([] ⊢m s <-> [] ⊢c s).
```

**h.** Deduce that minimal natural deduction is consistent if and only if classical natural deduction is consistent, in other words that one proves $\perp$ if and only if the other does.

## 2 Hilbert Systems and combinatory logic

We will now work towards a consistency proof of minimal natural deduction. There is more than one approach, we here choose to use an approach based on proof terms and strong normalisation. Mechanising proof terms and normalisation proofs can be hard, mainly because they need to deal with substitution of variables. We circumvent this by introducing Hilbert Systems (independently introduced by Frege and Hilbert). They are a way to present deduction without having to manage the context. Consequently, their proof terms will not require substitution, leading to a rather elegant strong normalisation proof.

### 2.1 Hilbert Systems

We give the rules for the Hilbert systems below. While there is a context to be able to do global axiomatic assumptions, the context never changes during a proof.

$$\frac{s \in A}{A \vdash_H s} \qquad \frac{A \vdash_H s \to t \qquad A \vdash_H s}{A \vdash_H t} \qquad \frac{}{A \vdash_H s \to t \to s}$$

$$\frac{}{A \vdash_H (s \to t \to u) \to (s \to t) \to s \to u}$$

**a.** Define an inductive predicate `hil : list form -> form -> Prop` capturing the rules from above. Make sure you turn the first argument into a *parameter*. Declare the notation `A ⊢H s` for `hil A s`.

**b.** Prove that Hilbert provability implies minimal provability, *i.e.*

```
Lemma hil_ndm A s :
  hil A s -> ndm A s.
```

**c.** Show the following 3 facts:

1. If $A \vdash_H s$ then $A \vdash_H t \to s$.
2. If $A \vdash_H s \to t \to u$ and $A \vdash_H s \to t$ then $A \vdash_H s \to u$.
3. $A \vdash_H s \to s$.

**d.** Prove that if $s :: A \vdash_H t$ then $A \vdash_H s \to t$. Explain briefly why it is crucial that $A$ is a parameter of the `hil` predicate.

**e.** Prove that minimal provability implies Hilbert provability, *i.e.*

```
Fact ndm_hil {A s} :
  ndm A s -> hil A s.
```

## 2.2 Abstract reduction systems

We work abstractly with a reduction relation $R : A \to A \to \mathsf{Prop}$. In Coq, use a section as follows:

```
Require Import Lia ZArith List.
equire Import ex1.

Section ARS.

  Context [A : Type]. Variable R : A -> A -> Prop.
```

Such a relation is strongly normalising on an element $x : A$ if all reduction paths from $x$ are finite. We can define this notion inductively as follows:

$$\frac{\forall y.\ R\ x\ y \to \mathsf{SNon}\ R\ y}{\mathsf{SNon}\ R\ x}$$

**a.** Define an inductive relation `SN_on : A -> Prop` with the above rules. Work in the section above.

**b.** Define the reflexive transitive closure of $R$, mathematically defined as follows

$$\frac{}{R^*\ x\ x} \qquad \frac{R\ x\ y}{R^*\ x\ y} \qquad \frac{R^*\ x\ y \qquad R^*\ y\ z}{R^*\ x\ z}$$

as an inductive relation `rtc : A -> A -> Prop`.

**c.** Prove

```
Lemma SN_on_rtc x y :
  SN_on x -> rtc x y -> SN_on y.
```

**d.** Given a typing relation $T : A \to \mathsf{Prop}$ and a value relation $V : A \to \mathsf{Prop}$, strong normalisation implies the existence of normal forms if $R$ preserves typing and satisfies progress, *i.e.* any well-typed term, either steps or is a value. Formally:

```
Variables T V : A -> Prop.

Variable Hpres : forall x y, T x -> R x y -> T y.
Variable Hprog : forall x, T x -> (exists y, R x y) \/ V x.

Lemma SN_to_WN x :
  T x -> SN_on x -> exists v, rtc x v /\ T v /\ V v.
```

**e.** You can now close the ARS section with `End ARS`. Prove double induction on strong normalisation proofs, *i.e.*

```
Lemma SN_on_double_ind [A B : Type] [R1 : A -> A -> Prop] [R2 : B -> B -> Prop]
          (P : A -> B -> Prop) :
  (forall (a : A) (b : B),
    (forall (a' : A), R1 a a' -> SN_on R1 a') ->
    (forall (a' : A), R1 a a' -> P a' b) ->
    (forall (b' : B), R2 b b' -> SN_on R2 b') ->
    (forall (b' : B), R2 b b' -> P a b') ->
    P a b) ->
  forall (x : A) (y : B), SN_on R1 x -> SN_on R2 y -> P x y.
```

## 2.3 Combinatory Logic

Combinatory logic was introduced by Schoenfinkel and Curry. It can be seen as exactly the proof terms of a Hilbert system.

```
Inductive term :=
| S | K | V (n : nat) | app (e1 e2 : term).
Coercion app : term >-> Funclass.

(* Import your solution to exercise 1. *)
From Project Require Import ex1.

Implicit Type s t : form.
Implicit Type e : term.

Ltac inv H := inversion H; subst; clear H.

Section typing.

  Variable A : list form.

  Reserved Notation "⊢ e : s" (at level 60, e at next level).
```

We write $e_1\ e_2$ for app $e_1\ e_2$, and you can do it in your code too thanks to the `Coercion` line.

**a.** Define a typing relation `typing : term -> form -> Prop` as follows:

$$\frac{\mathsf{nth\_error}\ A\ n = \mathsf{Some}\ s}{A \vdash \mathsf{V}\ n : s} \qquad \frac{A \vdash e_1 : s \to t \qquad A \vdash e_2 : s}{A \vdash e_1\ e_2 : t}$$

$$\frac{}{A \vdash \mathsf{K} : s \to t \to s} \qquad \frac{}{A \vdash \mathsf{S} : (s \to t \to u) \to (s \to t) \to s \to u}$$

with notation `Notation "⊢ e : s" := (typing e s) (at level 60, e at next level).`

**b.** Show that Hilbert system provability and the existence of well-typed proof terms are equivalent:

```
Lemma hil_equiv s :
  hil A s <-> exists e, ⊢ e : s.
```

**c.** Define a reduction relation `red : term -> term -> Prop` as follows:

$$\frac{}{\mathsf{K}\ e_1\ e_2 \succ e_1} \qquad \frac{}{\mathsf{S}\ e_1\ e_2\ e_3 \succ e_1\ e_3\ (e_2\ e_3)} \qquad \frac{e_1 \succ e_1'}{e_1\ e_2 \succ e_1'\ e_2} \qquad \frac{e_2 \succ e_2'}{e_1\ e_2 \succ e_1\ e_2'}$$

with notation `Notation "e1 ≻ e2" := (red e1 e2) (at level 60).`

**d.** Show that one step reduction preserves types, *i.e.*

```
Lemma preservation e1 e2 s :
  ⊢ e1 : s ->
  e1 ≻ e2 ->
  ⊢ e2 : s.
```

**e.** We define the reflexive transitive closure of reduction as follows.

```
Definition reds :=
  rtc red.

Notation "e1 ≻* e2" := (reds e1 e2) (at level 60).
```

Prove the following congruence lemma for reduction and application:

```
Lemma app_red e1 e1' e2 :
  e1 ≻* e1' ->
  e1 e2 ≻* e1' e2.
```

**f.** Prove that type preservation also extends to the reflexive transitive closure of reduction, *i.e.*

```
Lemma subject_reduction e1 e2 s :
  ⊢ e1 : s ->
  e1 ≻* e2 ->
  ⊢ e2 : s.
```

**g.** We now define strongly normalising terms and prove that if an application is strongly normalising, then so is the left-hand term. We close the section first.

```
End typing.

Notation "A ⊢ e : s" := (typing A e s) (at level 60, e at next level).

Notation "t1 ≻ t2" := (red t1 t2) (at level 60).
Notation "t1 ≻* t2" := (reds t1 t2) (at level 60).

Definition SN (e : term) :=
  SN_on red e.

Lemma SN_app e1 e2 :
  SN (e1 e2) -> SN e1.
```

**h.** We now define so-called neutral terms and show they are closed under application:

```
Definition neutral (e : term) :=
  match e with
  | app K _ | K | app (app S _) _ | S | app S _ => False
  | _ => True
  end.

Lemma neutral_app e1 e2 :
  neutral e1 -> neutral (e1 e2).
```

**i.** Finally, we prove that well-typed terms either step or are some form of value. In this case, we can define a term to be a value if it is not neutral:

```
Lemma progress e s :
  (nil ⊢ e : s) -> (exists e', red e e') \/ ~neutral e.
```

## 2.4 Normalisation

For this exercise, we give a detailed proof on paper. Your task is to formalise all notions in Coq, and prove the lemmas and theorems. You will not have to invent mathematical arguments or intermediate lemmas: the structure of the proof is given on paper without gaps.

**Definition 1.** *We define a notion of semantic typing* $\vDash e : s$ *as a recursive function on* $s$*:*

$$\vDash e : \bot := \mathsf{SN}\ e \qquad \vDash e : \mathsf{var}\ x := \mathsf{SN}\ e \qquad \vDash e : s_1 \to s_2 := \forall e_1.\ \vDash e_1 : s_1 \to \vDash e\ e_1 : s_2$$

**Theorem 1.** *The following holds for all* $s$ *and* $e$*:*

1. *If* $\vDash e : s$ *then* $\mathsf{SN}\ e$.
2. *If* $\vDash e : s$ *then for all* $e'$ *with* $e \succ^* e'$ *it holds that* $\vDash e' : s$.
3. *If* $e$ *is neutral and for all* $e'$ *with* $e \succ e'$ *we have* $\vDash e' : s$*, it holds that* $\vDash e : s$.

*Proof.* By induction on $s$ with $e$ generalised.

We have three cases, but the proof is the same for $s = \mathsf{var}\ x$ and $s = \bot$. (1) is trivial. (2) follows from `SN_on_rtc`. (3) is by definition of strong normalisation.

Now let $s = s_1\ s_2$. We have three induction hypotheses each for $s_1$ and $s_2$ and three things to prove.

For (1), we can assume that $e$ semantically has type $s_1 \to s_2$, *i.e.* that for all $e_1$ with $\vDash e_1 : s_1$ we have $\vDash e\ e_1 : s_2$. We need to prove that $e$ strongly normalises. We use `SN_app` with $e_2 := \mathsf{V}\ 0$ (or any other variable), so we have to prove that $\mathsf{SN}\ (e\ (\mathsf{V}\ 0))$. We use the induction hypothesis (1) for $s_2$ and have to prove $\vDash e\ (\mathsf{V}\ 0) : s_2$. By assumption, it suffices to prove $\vDash \mathsf{V}\ 0 : s_1$. We use the induction hypothesis (3) for $s_1$. $\mathsf{V}\ 0$ is neutral and does not reduce to anything, so we are done.

For (2), assume that $e$ semantically has type $s_1 \to s_2$, *i.e.* that for all $e_1$ with $\vDash e_1 : s_1$ we have $\vDash e\ e_1 : s_2$ and that $e \succ^* e'$. We need to prove that $e'$ semantically has type $s_1 \to s_2$, *i.e.* assume $e_1$ with $\vDash e_1 : s_1$ and need to prove that $\vDash e'\ e_1 : s_2$. We use the induction hypothesis (2) for $s_2$ and that $\vDash e_1 : s_1$. It suffices to prove that $e\ e_1 \succ^* e'\ e_1$, which follows from `app_red`.

For (3), assume that $e$ is neutral and for all $e'$ with $e \succ e'$ we have $\vDash e' : s_1 \to s_2$ (call this assumption $\mathsf{H}$). We need to prove that $e$ semantically has type $s_1 \to s_2$, *i.e.* assume $e_1$ with $\vDash e_1 : s_1$ and need to prove that $\vDash e\ e_1 : s_2$. We use the induction hypothesis (1) for $s_1$ on the assumption to derive $\mathsf{SN}\ e_1$. Now the proof is by induction on this strong normalisation proof, *i.e.* we can assume that for any $e_1'$ with $e_1 \succ e_1'$ with $\vDash e_1' : s_1$ we have that $\vDash e\ e_1' : s_2$. We use the induction hypothesis (3) for $s_2$. We have to prove that $e\ e_1$ is neutral, which follows from $e$ being neutral from `neutral_app`, and then assume $e'$ with $ee_1 \succ e'$ and have to prove that $\vDash e' : s_2$. We do a case analysis on $e\ e_1 \succ e'$. There are four cases:

- For $e = \mathsf{K}\ e_3$, we have a contradiction because $\mathsf{K}\ e_3$ is not neutral.
- For $e = \mathsf{S}\ e_3\ e_4$, we have a contradiction because $\mathsf{S}\ e_4\ e_5$ is not neutral.
- For $e \succ e_3$, we can use $\mathsf{H}$ and are done.
- For $e_1 \succ e_1'$, we first use the induction hypothesis for $e_1$, and it suffices to prove that $\vDash e_1' : s_1$. This follows from using the induction hypothesis (2) for $s_1$, because $\vDash e_1 : s_1$ and $e_1 \succ^* e_1'$ follows from $e_1 \succ e_1'$.

□

**Lemma 2.** *For any $s$ and $t$, $\vDash \mathsf{K} : s \to t \to s$*

*Proof.* By definition, we can assume $\vDash e_1 : s$ and $\vDash e_2 : t$ and have to prove $\vDash \mathsf{K}\ e_1\ e_2 : s$. By application of Theorem 1, we know that $e_1$ and $e_2$ are strongly normalising. We use double induction on strong normalisation, proved above.

We use Theorem 1, point (3), to prove $\vDash \mathsf{K}\ e_1\ e_2 : s$. We need to show that $\mathsf{K}\ e_1\ e_2$ is neutral – which follows by definition, It remains to show that any $e'$ with $\mathsf{K}\ e_1\ e_2 \succ e'$ fulfills $\vDash e' : s$. There are three cases:

1. $\mathsf{K}\ e_1\ e_2 \succ e_1$. $\vDash e_1 : s$ follows by assumption.
2. $\mathsf{K}\ e_1\ e_2 \succ e'\ e_2$ via $\mathsf{K}\ e_1 \succ e'$. This cannot arise from $\mathsf{K}$ stepping, so it has to be that $e' = \mathsf{K}\ e_1'$ and $e_1 \succ e_1'$. We have to prove $\vDash \mathsf{K} e_1'\ e_2$.

   We can apply the induction hypothesis for $e_1$ with $e_1 \succ e_1'$. We need to prove $\vDash e_1' : s$, which follows from Theorem 1 point (2).
3. $\mathsf{K}\ e_1\ e_2 \succ \mathsf{K}\ e_1\ e_2'$ via $e_1' \succ e_2'$.

   We can apply the induction hypothesis for $e_2$ with $e_2 \succ e_2'$. We need to prove $\vDash e_2' : t$, which follows from Theorem 1 point (2).

□

**Lemma 3.** *For any $s$, $t$, $u$, $\vDash \mathsf{S} : (s \to t \to u) \to (s \to t) \to s \to u$.*

*Proof.* Essentially the same proof as for $\mathsf{K}$. But one first needs to state and prove a *triple* induction principle for strong normalisation. □

We recommend proving the lemma about $\mathsf{S}$ *last*, *i.e.* stating and admitting it first.

**Theorem 4.** *Assume that whenever the $n$-th element of $A$ is $s$, $\vDash \mathsf{V}\ n : s$ holds. Then $A \vdash e : s$ implies $\vDash e : s$.*

*Proof.* By induction on typing. The variable case uses the assumption. The $\mathsf{K}$ and $\mathsf{S}$ cases follow from the last two lemmas. The application case follows from the induction hypotheses. □

**Lemma 5.** *Any term well typed in the empty context is strongly normalising.*

*Proof.* Straightforward from the last two theorems. □

**Lemma 6.** *Any well-typed term $e$ in the empty context reduces to a term $e'$ of the same type which is not neutral.*

*Proof.* Follows from `SN_to_WN`. □

## 2.5 Consistency

We are going to prove that the Hilbert system is consistent by proving that there is no normal term of type $\perp$.

**a.** Prove that there is no term of type `bot` in the empty context.

```
Lemma noterm e :
  ~[]  ⊢ e : bot.
```

Use weak normalisation first and then do case analysis on the proof that the value is not neutral until the goal is proved.

**b.** Prove that intuitionistic natural deduction is consistent, *i.e.*

```
Corollary nd_consistent :
  ∼ []  ⊢m bot.
```

**c.** Prove that classical natural deduction is consistent, *i.e.*

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

## 3 Mini projects to choose

Choose one or several of the following mini projects to obtain more than 16 points on the project. Submit them in separate files, together with explanations of what you decided to do in comments at the beginning of the files.

### 3.1 More on logic and normalisation

- 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 truth with a unit type as proof terms (easy), conjunction with pairs as proof terms (relatively easy), or disjunctions with sums as proof terms (less easy), then re-prove all theorems. Please submit a separate file for this.
- Prove that classical natural deduction is decidable (*e.g.* via a sound and complete boolean semantics).
- Introduce simply typed $\lambda$-calculus as proof terms for minimal logic. Show that type-checking simply typed $\lambda$-terms is decidable.
- (Hard) Define a reduction relation for simply typed $\lambda$ calculus, and show that $\lambda$-calculus can be simulated using combinatory logic. Ask as for advice before going this route.

### 3.2 Reflection

The goal is to write an automatic tactic for proving that two Boolean 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 $\mathbb{Z}$:

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

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))).
```