

Provability and parametricity for natural deduction

v1.0, 20 October

Please hand in a zip file containing the five exercises called `ex1.v`, `ex2.v`, `ex3.v`, `ex4.v` and `ex5.v`, as well as a `README.txt` file explaining how to build and navigate your proofs. **The subject of your email should be “[MPRI PRFA] Project”** or you risk it being missed by us. Our emails: `yannick.forster@inria.fr` and `theo.winterhalter@inria.fr`.

Asking for help. You are encouraged to ask questions about the project on Discord.

You are allowed to discuss the project with your fellow students. You are *not* allowed to share code in any way: copy-pasting, sending it to others, or looking at others’ code is forbidden.

Please start your `README.txt` file with a summary of who you discussed what aspect with. This includes both questions you ask fellow students *and* help you give to fellow students. An example for this is:

- I asked X about whether induction on the formula is needed in exercise 4.
- I was asked by Y whether I defined the predicate in exercise 1.2.b using **Fixpoint** or **Inductive**.

You are not allowed to use generative AI / large language models (LLMs) such as ChatGPT or Claude for any aspect of the project.

README. Submit a `README.txt` file where you

- explain how to build the project,
- explain who you discussed the project with,
- explain your previous experience with Rocq or other proof assistants,
- state which exercises you did not solve and explain why you got stuck.

Building projects. Ensure that your project builds with Rocq 9.2. You are allowed to use the Equations package. Please ask us if you intend to use anything else. To help us read your project, please identify which question you are answering by using comments with question numbers such as `(* 1.2.b *)`.

Comments. Please add comments to your Rocq file explaining design choices and difficulties you encountered when they do not fit in the README.

Evaluation. By solving exercises 1 to 3, you will get at least 12 points out of 20.

Advice. Take a step back whenever you are stuck. Doing Rocq 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.

Unicode. We use Unicode in this document to make it easier on the eyes, but you are free not to use it in your project. Remember that we provide advice about Unicode in our FAQ.

Changes. We will publish new versions of this PDF in case it becomes necessary, *e.g.* 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.

Deadline: 19 November 2026 at 18:00.

Late submissions are only accepted if you have a good reason, in which case, please talk to us before November 12th.

1 Propositional logic

In this exercise, we will define a natural deduction system for propositional logic. We will in fact study *minimal logic* which does not feature the principle of explosion (from false, everything follows).

1.1 Natural deduction

Mathematically, the natural deduction system we consider has 3 rules (assumption, implication introduction, and implication elimination):

$$\frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{A :: \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

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

```
From Stdlib Require Import List.
Import ListNotations.

Inductive form : Type :=
| fVar (n : nat) | fImp (A B : form).

Print In.
Print incl.

Abbreviation  $\alpha$  := fVar.
Notation " $A \Rightarrow B$ " := (fImp A B) (at level 51, right associativity).
Abbreviation bot := ( $\alpha$  0).
Abbreviation neg A := (A  $\Rightarrow$  bot).
Reserved Notation " $\Gamma \vdash A$ " (at level 70, A at next level).
```

- a. Define an inductive predicate `nd : list form  $\rightarrow$  form  $\rightarrow$  Prop` capturing the rules from above. Declare the notation $\Gamma \vdash A$ for `nd  $\Gamma$  A`. Herein, Γ is what we call the *context*.
- b. Construct natural deduction proofs of the following statements:
 1. $\Gamma \vdash A \Rightarrow A$,
 2. $A :: \Gamma \vdash \text{neg } (\text{neg } A)$,
 3. $\text{neg } (\text{neg } \text{bot}) :: \Gamma \vdash \text{bot}$.

- c. Prove weakening:

```
Fact Weak  $\Gamma \Delta A$  :
   $\Gamma \vdash A \rightarrow \text{incl } \Gamma \Delta \rightarrow \Delta \vdash A$ .
```

1.2 Model-based semantics

We are now going to build models of natural deduction which will help us deduce for instance consistency. In this exercise, we're going to use the following definition of model:

```
Definition model := nat  $\rightarrow$  Prop.
```

- a. Define a function `interp : model → form → Prop` that takes `M : model` as argument to interpret variables. The definition we want is of the following form:

$$\begin{aligned} \llbracket \text{fVar } n \rrbracket_M &:= M(n) \\ \llbracket A \Rightarrow B \rrbracket_M &:= \llbracket A \rrbracket_M \rightarrow \llbracket B \rrbracket_M \end{aligned}$$

where we write $\llbracket \cdot \rrbracket_M$ for `interp M`.

- b. We now extend this definition to contexts as follows:

$$\begin{aligned} \llbracket [] \rrbracket_M &:= \top \\ \llbracket A :: \Gamma \rrbracket_M &:= \llbracket A \rrbracket_M \wedge \llbracket \Gamma \rrbracket_M \end{aligned}$$

Implement this as a function `ctx_interp : model → list form → Prop`.

We define *validity* of a formula `A` in the model `M` as `interp M A` and use the following notation.

Notation `"M ⊨ A"` := (`interp M A`) (**at level** 60, **A at next level**).

- c. We are now going to prove consistency. To that end, we first prove the following soundness lemma of natural deduction with respect to model-based semantics:

Lemma `interp_sound M Γ A :`
`Γ ⊢ A →`
`ctx_interp M Γ →`
`M ⊨ A.`

- d. Deduce consistency from soundness. Consistency is expressed as the fact that there exists at least one underivable formula in the empty context. We pick here `fVar 0` which we conveniently used already in our notion of negation.

Lemma `consistency :`
`¬ [] ⊢ fVar 0.`

- e. Prove that if a model is classical, it validates Peirce's law:

Lemma `peirce_valid M A B :`
`(∀ n : nat, M n ∨ ¬ M n) →`
`M ⊨ ((A ⇒ B) ⇒ A) ⇒ A.`

1.3 Hilbert systems

We now introduce Hilbert systems, a presentation of deduction which does not require any context manipulation.

$$\frac{A \in \Gamma}{\Gamma \vdash_H A} \quad \frac{\Gamma \vdash_H A \Rightarrow B \quad \Gamma \vdash_H A}{\Gamma \vdash_H B} \quad \frac{}{\Gamma \vdash_H A \Rightarrow B \Rightarrow A}$$

$$\frac{}{\Gamma \vdash_H (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C}$$

- 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 $\Gamma \vdash_H A$ for `hil Γ A`.

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

Lemma `hil_nd` ΓA :
 $\Gamma \vdash_H A \rightarrow \Gamma \vdash A$.

- c. Show the following 3 facts:

1. If $\Gamma \vdash_H A$ then $\Gamma \vdash_H B \Rightarrow A$.
2. If $\Gamma \vdash_H A \Rightarrow B \Rightarrow C$ and $\Gamma \vdash_H A \Rightarrow B$ then $\Gamma \vdash_H A \Rightarrow C$.
3. $\Gamma \vdash_H A \Rightarrow A$.

- d. Prove that if $A :: \Gamma \vdash_H B$ then $\Gamma \vdash_H A \Rightarrow B$. Explain briefly why it is crucial that Γ is a parameter of the `hil` predicate.

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

Fact `nd_hil` ΓA :
 $\Gamma \vdash A \rightarrow \Gamma \vdash_H A$.

2 Proof terms

We will now introduce proof terms for natural deduction. Mechanising proof terms and normalisation proofs can be hard, mainly because they need to deal with substitution of variables. Luckily, the proof terms of Hilbert systems will not require substitution, leading to a rather elegant development.

2.1 Combinatory Logic

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

Start a file `ex2.v` as follows. To import your solution to exercise 1, compile your files with a `_CoqProject` file containing the line `-Q . Project`.

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

Inductive term :=
| tVar (n : nat) | tApp (s t : term) | tK | tS.

Notation "s · t" := (tApp s t) (at level 20, left associativity).

Definition intf := list form.

Reserved Notation "Γ ⊢ t : A" (at level 70, t at next level).
Reserved Notation "s > t" (at level 60).
```

We write $s \cdot t$ for `tApp s t`, and you can do it in your code too thanks to the notation. We call lists of formulas used as typing contexts *interfaces*, hence the name `intf`.

- a. Define a typing relation `typing : intf → term → form → Prop` as follows:

$$\frac{\text{nth_error } \Gamma \ n = \text{Some } A}{\Gamma \vdash \text{tVar } n : A} \qquad \frac{\Gamma \vdash s : A \Rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash s \cdot t : B}$$

$$\frac{}{\Gamma \vdash \text{tK} : A \Rightarrow B \Rightarrow A} \qquad \frac{}{\Gamma \vdash \text{tS} : (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C}$$

with notation

```
Notation "Γ ⊢ t : A" := (typing Γ t A) (at level 70, t at next level).
```

Note that `nth_error` is provided by the standard library.

- b. We can define the identity as follows. Show it has the expected type.

```
Definition tI : term :=
  tS · tK · tK.
```

```
Lemma type_I Γ A :
  Γ ⊢ tI : A ⇒ A.
```

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

```
Lemma hil_typing Γ A :
  Γ ⊢H A ↔ ∃ t, Γ ⊢ t : A.
```

- d. Define a reduction relation $\text{red} : \text{term} \rightarrow \text{term} \rightarrow \text{Prop}$ as follows:

$$\frac{}{\text{tK} \cdot s \cdot t \succ s} \quad \frac{}{\text{tS} \cdot s \cdot t \cdot u \succ s \cdot u \cdot (t \cdot u)} \quad \frac{s \succ s'}{s \cdot t \succ s' \cdot t} \quad \frac{t \succ t'}{s \cdot t \succ s \cdot t'}$$

with notation

Notation " $s \succ t$ " := (red s t) (at level 60).

- e. Show that one-step reduction preserves typing, *i.e.*

Lemma preservation $\Gamma \vdash s \vdash A$:

$$\begin{array}{l} \Gamma \vdash s : A \rightarrow \\ s \succ t \rightarrow \\ \Gamma \vdash t : A. \end{array}$$

2.2 Abstract reduction systems

To reason about sequences of reduction steps, and later about normalisation, we now step back and work abstractly with a reduction relation $R : T \rightarrow T \rightarrow \text{Prop}$. In Rocq, use a section as follows:

Section ARS.

Context {T : Type} (R : T → T → Prop).

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

$$\frac{\forall y. R \ x \ y \rightarrow \text{SN_on } R \ y}{\text{SN_on } R \ x}$$

- a. Define an inductive predicate $\text{SN_on} : T \rightarrow \text{Prop}$ with the above rule. Work in the ARS section.
- b. Define the reflexive transitive closure of R , mathematically defined as follows

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

as an inductive relation $\text{rtc} : T \rightarrow T \rightarrow \text{Prop}$.

- c. Prove

Lemma SN_on_rtc $x \ y$:

$$\text{SN_on } x \rightarrow \text{rtc } x \ y \rightarrow \text{SN_on } y.$$

- d. Given a typing relation $\text{Ty} : T \rightarrow \text{Prop}$ and a value relation $\text{Val} : T \rightarrow \text{Prop}$, 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 Ty Val : T → Prop.

Variable Hpres : $\forall x \ y, \text{Ty } x \rightarrow R \ x \ y \rightarrow \text{Ty } y$.

Variable Hprog : $\forall x, \text{Ty } x \rightarrow (\exists y, R \ x \ y) \vee \text{Val } x$.

Lemma SN_to_WN x :

$$\text{Ty } x \rightarrow \text{SN_on } x \rightarrow \exists y, \text{rtc } x \ y \wedge \text{Ty } y \wedge \text{Val } y.$$

Prove this lemma.

- 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) :
  (∀ (a : A) (b : B),
    (∀ (a' : A), R1 a a' → SN_on R1 a') →
    (∀ (a' : A), R1 a a' → P a' b) →
    (∀ (b' : B), R2 b b' → SN_on R2 b') →
    (∀ (b' : B), R2 b b' → P a b') →
    P a b) →
  ∀ (x : A) (y : B), SN_on R1 x → SN_on R2 y → P x y.

```

2.3 Multi-step reduction

- a. We use the following notation for the reflexive transitive closure of reduction.

Notation " $s \succ^* t$ " := (rtc red s t) (at level 60).

Prove the following basic properties of multi-step reduction.

```

Lemma reds_step s t u :
  s > t →
  t >* u →
  s >* u.

```

```

Lemma reds_app_l s s' t :
  s >* s' →
  s · t >* s' · t.

```

```

Lemma reds_app_r s t t' :
  t >* t' →
  s · t >* s · t'.

```

- b. Prove the reduction rule for the identity.

```

Lemma tI_reds s :
  tI · s >* s.

```

- c. Type preservation extended to the reflexive transitive closure of reduction is often called *subject reduction*. Prove it.

```

Lemma subject_reduction Γ s t A :
  Γ ⊢ s : A →
  s >* t →
  Γ ⊢ t : A.

```

- d. Define a function `closed : term → Prop` which holds if there are no variables in the term.

- e. Prove that reduction preserves closedness:

```

Lemma closed_reds s t :
  s >* t →
  closed s →
  closed t.

```

- f. Prove that well-typed terms in the empty context are closed:

Lemma `closed_typing` t A :

$[] \vdash t : A \rightarrow$
`closed` t .

3 Logical relations

3.1 Semantic typing

We now introduce semantic typing, a generalisation of both typing and model-based semantics. Concretely, we make models parameterised by syntax, *i.e.*

Definition `pmodel := nat → term → Prop`.

Given such a model, P , we define validity for a term t and formula A , written $P \models t : A$, by induction on the formula as follows:

$$\begin{aligned} P \models t : \text{fVar } n &:= P(n)(t) \\ P \models f : A \Rightarrow B &:= \forall s. P \models s : A \Rightarrow P \models f \cdot s : B \end{aligned}$$

We now extend this definition to contexts:

$$P \models \Gamma := \forall n A. \Gamma[n] = \text{Some } A \rightarrow P \models \text{tVar } n : A$$

Note that this is harder to define inductively due to the reference to the index n .

We will now prove a property analogous to soundness for semantic typing, usually called the fundamental theorem.

- a. First, define validity `pinterp` and its extension to contexts `ctx_pinterp` in Rocq. You may use the following notation.

Notation `"P ⊨ t : A" := (pinterp P A t) (at level 60, t at next level)`.

- b. Prove that every model validates the empty context:

Lemma `ctx_pinterp_nil P :`
`ctx_pinterp P []`.

- c. Prove the following general form, saying that if `tK` and `tS` can be semantically typed, then every well-typed term can:

Section `fundamental`.

Variable `P : pmodel`.

Variable `hK : ∀ A B, P ⊨ tK : A ⇒ B ⇒ A`.

Variable `hS : ∀ A B C, P ⊨ tS : (A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C`.

Lemma `fundamental_gen Γ t A :`
`ctx_pinterp P Γ →`
`Γ ⊢ t : A →`
`P ⊨ t : A`.

Proof.

(Your proof here. *)*

Admitted.

End `fundamental`.

- d. We will now show a version of the fundamental lemma for all models closed under the converse of reduction, called *expansion*. Prove that if $P \models n$ is closed under expansion for every n , then so is validity `pinterp P A` for every A .

Definition `exp_closed` ($Q : \text{term} \rightarrow \text{Prop}$) :=
 $\forall s\ t, s \succ t \rightarrow Q\ t \rightarrow Q\ s.$

Lemma `exp_closed_pinterp` $P\ A :$
 $(\forall n, \text{exp_closed}\ (P\ n)) \rightarrow$
 $\text{exp_closed}\ (\text{pinterp}\ P\ A).$

- e. We introduce the closure of a predicate *up to* reduction. Prove the following properties.

Definition `upto` ($Q : \text{term} \rightarrow \text{Prop}$) $s :=$
 $\exists t, s \succ^* t \wedge Q\ t.$

Lemma `upto_in` ($Q : \text{term} \rightarrow \text{Prop}$) $s :$
 $Q\ s \rightarrow$
 $\text{upto}\ Q\ s.$

Lemma `exp_closed_upto` $Q :$
 $\text{exp_closed}\ (\text{upto}\ Q).$

- f. We can now deduce the following version of the fundamental lemma for predicates closed under expansion:

Lemma `fundamental` $P\ \Gamma\ t\ A :$
 $(\forall n, \text{exp_closed}\ (P\ n)) \rightarrow$
 $\text{ctx_pinterp}\ P\ \Gamma \rightarrow$
 $\Gamma \vdash t : A \rightarrow$
 $P \models t : A.$

3.2 Theorems for free

Coined by Wadler, the phrase “theorems for free” refers to properties that can be established of programs by simple virtue of being well typed. It comes from a rephrasing of Reynolds’ parametricity. The name refers to parametric polymorphism (the kind of polymorphism you see in OCaml), as opposed to ad-hoc polymorphism (often called overloading, present in languages like Python or C++) which may have different behaviour depending on the type.

Even without a notion of polymorphism, this idea is still relevant. In our case, we can show a myriad of properties by exploiting the fact that our type variables (`fVar n`) are unknown, and therefore that a program may not make assumptions about them.

To ease reading, we will now be using the abbreviation α for type variables (`fVar`) that we introduced in exercise 1.

- a. The first property we can establish is that to build a function from an unknown type to itself, the only possible way is to extensionally¹ build the identity function.

Lemma `free_id` $f\ s :$
 $[] \vdash f : \alpha\ 0 \Rightarrow \alpha\ 0 \rightarrow$
 $f \cdot s \succ^* s.$

- b. The same works for something with the type of `tK`.

Lemma `free_K` $f\ s\ t :$
 $[] \vdash f : \alpha\ 0 \Rightarrow \alpha\ 1 \Rightarrow \alpha\ 0 \rightarrow$
 $f \cdot s \cdot t \succ^* s.$

¹Here this means: when applied to an argument.

- c. We can use this semantics also to prove negative theorems. We show that our logic is not classical, concretely, that it does not derive Peirce's law:

Lemma `peirce_not_derivable` :
 $\neg [] \vdash ((\alpha\ 0 \Rightarrow \alpha\ 1) \Rightarrow \alpha\ 0) \Rightarrow \alpha\ 0.$

Hint: Make use of the fact that any `f` of this type would be closed.

- d. We can also use this idea to encode objects such as natural numbers and show they enjoy expected properties. For this we're going to pretend $\alpha\ 0$ is the type of natural numbers by considering a context in which one has some `tzero` : `Nat` and `tsucc` : `Nat` $\Rightarrow$ `Nat`. First we define the following:

Abbreviation `Nat` := ($\alpha\ 0$) (only parsing).
Abbreviation `tzero` := (`tVar` 0) (only parsing).
Abbreviation `tsucc` := (`tVar` 1) (only parsing).

Definition `INat` : `intf` := [
 (* `tzero` : *) `Nat` ;
 (* `tsucc` : *) `Nat` $\Rightarrow$ `Nat`
].

Start with proving that `tzero` and `tsucc` have the expected types:

Lemma `type_zero` :
`INat` $\vdash$ `tzero` : `Nat`.

Lemma `type_succ` :
`INat` $\vdash$ `tsucc` : `Nat` $\Rightarrow$ `Nat`.

- e. Now, prove that these natural numbers enjoy the usual induction principle:

Lemma `nat_induction` `Q` `t` :
`Q` `tzero` $\rightarrow$
 $(\forall\ s,\ Q\ s \rightarrow Q\ (tsucc\ \cdot\ s)) \rightarrow$
`INat` $\vdash$ `t` : `Nat` $\rightarrow$
`upto` `Q` `t`.

- f. We can also show that every term of type `Nat` is equivalent to a numeral:

Inductive `numeral` : `term` $\rightarrow$ **Prop** :=
| `num_z` : `numeral` `tzero`
| `num_s` `t` : `numeral` `t` $\rightarrow$ `numeral` (`tsucc` $\cdot$ `t`).

Lemma `nat_numeral` `t` :
`INat` $\vdash$ `t` : `Nat` $\rightarrow$
`upto` `numeral` `t`.

3.3 Stronger theorems for free

We can prove even more properties if we strengthen our fundamental theorem. Until now, we considered terms in our context that remain abstract: `tzero` is always `tzero`, in other words a variable. But variables are meant to be instantiated – or substituted. If we see this as a form of (ML) module abstraction, then it becomes natural to provide a concrete implementation for the module. Allowing instantiation in the fundamental theorem lets us derive stronger theorems.

We'll look at the Booleans now to understand the limitations and then to generalise the theorem.

- a. We use the following notations (similar to what we did for natural numbers).

Abbreviation Bool := (α 0) (only parsing).
Abbreviation ttrue := (tVar 0) (only parsing).
Abbreviation tfalse := (tVar 1) (only parsing).
Abbreviation tif := (tVar 2) (only parsing).

Definition IBool : intf := [
 (* ttrue : *) Bool ;
 (* tfalse : *) Bool ;
 (* tif : *) Bool $\Rightarrow$ Bool $\Rightarrow$ Bool $\Rightarrow$ Bool
].

Prove typing for ttrue, tfalse and tif.

- b. Compared with natural numbers, we also included the eliminator in the context (or interface). Given a term t of type Bool – i.e. such that $\text{IBool} \vdash t : \text{Bool}$, the fundamental theorem isn't enough to conclude that t reduces to either ttrue or tfalse. Provide an irreducible t of type Bool that is neither. Include proofs for all three: typing, irreducibility and inequality with ttrue and tfalse.
- c. We will improve the situation by providing a concrete implementation for tif which *computes* depending on the value of the Boolean passed to it. First, we need to define this notion of instantiation. Define a function $\text{inst} : (\text{nat} \rightarrow \text{term}) \rightarrow \text{term} \rightarrow \text{term}$ such that $\text{inst } \sigma \ t$ is t where tVar n is replaced by $\sigma \ n$.
- d. Show that the identity substitution results in the identity for instantiation:

Lemma inst_id s :
 inst tVar s = s .

- e. We now update our notion of validity of a context Γ in a model: a model P and a substitution $\sigma : \text{nat} \rightarrow \text{term}$ *implement* a context Γ if for every variable n of type A in Γ we have $P \models \sigma \ n : A$. Define this as $\text{pimplements } P \ \sigma \ \Gamma$.
- f. As before, prove a first version of the fundamental theorem.

Lemma fundamental_inst_gen $P \ \Gamma \ t \ A \ \sigma$:
 ($\forall A \ B, P \models tK : A \Rightarrow B \Rightarrow A$) $\rightarrow$
 ($\forall A \ B \ C, P \models tS : (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$) $\rightarrow$
 pimplements $P \ \sigma \ \Gamma \rightarrow$
 $\Gamma \vdash t : A \rightarrow$
 $P \models \text{inst } \sigma \ t : A$.

- g. Deduce the new fundamental theorem.

Lemma fundamental_inst $P \ \Gamma \ t \ A \ \sigma$:
 ($\forall n, \text{exp_closed } (P \ n)$) $\rightarrow$
 pimplements $P \ \sigma \ \Gamma \rightarrow$
 $\Gamma \vdash t : A \rightarrow$
 $P \models \text{inst } \sigma \ t : A$.

- h. Define an implementation for the Boolean interface: $\text{IBool_impl} : \text{nat} \rightarrow \text{term}$. We implement ttrue as tK, tfalse as tK · tI, and tif as tI. It doesn't matter how the other variables are interpreted.
- i. Prove the following auxiliary lemma.

```

Lemma upto_reduces (Q : term → Prop) s t :
  s >= t →
  upto Q t →
  upto Q s.

```

- j. We can finally prove our lemma that any Boolean reduces to the implementation of either `ttrue` or `tfalse`.

```

Lemma bool_impl s :
  IBool ⊢ s : Bool →
  inst IBool_impl s >= tK ∨ inst IBool_impl s >= tK · tI.

```

4 Normalisation

- a. We define what it means for a term to be strongly normalising, and prove that if an application is strongly normalising, then so is the left-hand term.

Definition `SN (s : term) :=
 SN_on red s.`

Lemma `SN_app s t :
 SN (s · t) → SN s.`

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

Definition `neutral (s : term) :=
 match s with
 | tK | tK · _ | tS | tS · _ | tS · _ · _ => False
 | _ => True
 end.`

Lemma `neutral_app s t :
 neutral s → neutral (s · t).`

- c. 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 t A :
 ([] ⊢ t : A) → (exists t', t > t') ∨ ~ neutral t.`

Normalisation, the pen-and-paper way. For this exercise, we give a detailed proof on paper. Your task is to formalise all notions in Rocq, 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.

We will make use of the semantic typing relation with interpretation $P \Vdash t := \text{SN } t$.

Theorem 1. *The following hold for all A and t :*

1. *If $P \Vdash t : A$ then $\text{SN } t$.*
2. *If $P \Vdash t : A$ then for all t' with $t > t'$ then we have $P \Vdash t' : A$.*
3. *If t is neutral and for all t' with $t > t'$ we have $P \Vdash t' : A$, then it holds that $P \Vdash t : A$.*

Proof. By induction on A , generalising over t .

For $A = \text{fVar } n$: (1) is trivial. (2) and (3) are by definition of strong normalisation.

Now let $A = A_1 \Rightarrow A_2$ and write f for t . We have three induction hypotheses each for A_1 and A_2 and three things to prove.

For (1), we can assume that f semantically has type $A_1 \Rightarrow A_2$, i.e. that for all s with $P \Vdash s : A_1$ we have $P \Vdash f \cdot s : A_2$. We need to prove that f strongly normalises. We use `SN_app` with $t := \text{tVar } 0$ (or any other variable), so we have to prove that $\text{SN } (f \cdot \text{tVar } 0)$. We use the induction hypothesis (1) for A_2 and have to prove $P \Vdash f \cdot \text{tVar } 0 : A_2$. By assumption, it suffices to prove $P \Vdash \text{tVar } 0 : A_1$. We use the induction hypothesis (3) for A_1 . `tVar 0` is neutral and does not reduce to anything, so we are done.

For (2), assume that f semantically has type $A_1 \Rightarrow A_2$, *i.e.* that for all s with $P \models s : A_1$ we have $P \models f \cdot s : A_2$ and that $f \succ f'$. We need to prove that f' semantically has type $A_1 \Rightarrow A_2$, *i.e.* assume s with $P \models s : A_1$ and need to prove that $P \models f' \cdot s : A_2$. We use the induction hypothesis (2) for A_2 and that $P \models s : A_1$. It suffices to prove that $f \cdot s \succ f' \cdot s$, which follows from $f \succ f'$.

For (3), assume that f is neutral and for all f' with $f \succ f'$ we have $P \models f' : A_1 \Rightarrow A_2$ (call this assumption H). We need to prove that f semantically has type $A_1 \Rightarrow A_2$, *i.e.* assume s with $P \models s : A_1$ and need to prove that $P \models f \cdot s : A_2$. We use the induction hypothesis (1) for A_1 on the assumption $P \models s : A_1$ to derive SN s . Now the proof is by induction on this strong normalisation proof, *i.e.* we can assume that for any s' with $s \succ s'$ and $P \models s' : A_1$ we have that $P \models f \cdot s' : A_2$. We use the induction hypothesis (3) for A_2 . We have to prove that $f \cdot s$ is neutral, which follows from f being neutral by `neutral_app`, and then assume t' with $f \cdot s \succ t'$ and have to prove that $P \models t' : A_2$. We do a case analysis on $f \cdot s \succ t'$. There are four cases:

- For $f = \mathbf{tK} \cdot u$, we have a contradiction because $\mathbf{tK} \cdot u$ is not neutral.
- For $f = \mathbf{tS} \cdot u \cdot u'$, we have a contradiction because $\mathbf{tS} \cdot u \cdot u'$ is not neutral.
- For $f \succ f'$, we can use H and are done.
- For $s \succ s'$, we first use the induction hypothesis for s , and it suffices to prove that $P \models s' : A_1$. This follows from using the induction hypothesis (2) for A_1 , because $P \models s : A_1$ and $s \succ s'$.

□

Lemma 2. *For any A and B , $P \models \mathbf{tK} : A \Rightarrow B \Rightarrow A$.*

Proof. By definition, we can assume $P \models s : A$ and $P \models t : B$ and have to prove $P \models \mathbf{tK} \cdot s \cdot t : A$. By application of Theorem 1, we know that s and t are strongly normalising. We use double induction on strong normalisation, proved above.

We use Theorem 1, point (3), to prove $P \models \mathbf{tK} \cdot s \cdot t : A$. We need to show that $\mathbf{tK} \cdot s \cdot t$ is neutral – which follows by definition. It remains to show that any u with $\mathbf{tK} \cdot s \cdot t \succ u$ fulfills $P \models u : A$. There are three cases:

1. $\mathbf{tK} \cdot s \cdot t \succ s$. $P \models s : A$ follows by assumption.
2. $\mathbf{tK} \cdot s \cdot t \succ u \cdot t$ via $\mathbf{tK} \cdot s \succ u$. This cannot arise from `tK` stepping, so it has to be that $u = \mathbf{tK} \cdot s'$ and $s \succ s'$. We have to prove $P \models \mathbf{tK} \cdot s' \cdot t : A$.

We can apply the induction hypothesis for s with $s \succ s'$. We need to prove $P \models s' : A$, which follows from Theorem 1 point (2).

3. $\mathbf{tK} \cdot s \cdot t \succ \mathbf{tK} \cdot s \cdot t'$ via $t \succ t'$.

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

□

Lemma 3. *For any A, B, C , $P \models \mathbf{tS} : (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$.*

Proof. Essentially the same proof as for `tK`. But one first needs to state and prove a *triple* induction principle for strong normalisation. □

We recommend proving the lemma about **tS** *last*, *i.e.* stating and admitting it first.

Lemma 4. *Any well-typed term is strongly normalising.*

Proof. We apply Theorem 1, point (1), and the generic fundamental lemma. The cases about **tK** and **tS** follow by Lemmas 2 and 3. The remaining case, validity of the context, requires Theorem 1, point (3): variables are neutral and do not reduce. $\square$

Lemma 5. *Any well-typed term t in the empty context reduces to a term t' of the same type which is not neutral.*

Proof. Follows from **SN_to_WN**, using **preservation** and **progress**. $\square$

5 LCF

Note. Start this exercise in a new file `ex5.v`.

We have shown one approach to using types to implement logic, via proof terms.

There is a second approach that does not rely on proof terms: LCF (logic of computable functions), introduced by Robin Milner. It is what powers proof assistants of the HOL family (HOL Light, HOL4, Isabelle/HOL).

LCF when implemented in a language like OCaml defines a module type as follows:

```
module type LCF = sig
  type thm
  val mp : thm → thm → thm
  val axK : form → form → thm
  val axS : form → form → form → thm
  val concl : thm → form
end
```

The idea is that the type `thm` is kept abstract, *i.e.* it can only be accessed through operations on it, but eventually `thm` is to be implemented literally as `form`. We have that *e.g.* $\text{concl } (\text{axK } A \ B) = A \Rightarrow B \Rightarrow A$ and $\text{concl } t = A \Rightarrow B \rightarrow \text{concl } t' = A \rightarrow \text{concl } (\text{mp } t \ t') = B$.

5.1 LCF in Rocq

- a. Implement a Rocq function `modus_ponens : form → form → form` such that

1. $\Gamma \vdash A \rightarrow \Gamma \vdash B \rightarrow \Gamma \vdash \text{modus_ponens } A \ B$
2. $\text{modus_ponens } (A \Rightarrow B) \ A = B$.

- b. Define an element `LCF_impl` of the following record:

```
Record LCF_intf := {
  thm : Type ;
  mp : thm → thm → thm ;
  axK : form → form → thm ;
  axS : form → form → form → thm ;
  concl : thm → form
}.
```

- c. Define a function `thmI : ∀ (A : form) (L : LCF_intf), thm L` such that

```
Lemma thmI_concl A :
  concl LCF_impl (thmI A LCF_impl) = A ⇒ A.
```

5.2 A calculus to prove LCF correct

The correctness of this implementation of LCF relies on the fact that Rocq itself is parametric. We cannot prove it correct.

If we want an implementation of LCF that is correct, we have to implement it on top of a typed calculus such as the combinatory logic we already saw.

We will do this, where we define the calculus to have a native type of formulas to work on.

```

Inductive type :=
| tyVar (n : nat) | tyForm | tyImp (T T' : type).

Abbreviation  $\alpha$  := tyVar.
Notation "T  $\Rightarrow$  T'" := (tyImp T T') (at level 51, right associativity).
Notation "A f $\Rightarrow$  B" := (fImp A B) (at level 51, right associativity).

Inductive term :=
| tVar (n : nat) | tApp (s t : term) | tK | tS
| tFVar (n : nat) | tImp | tMP.

Notation "s · t" := (tApp s t) (at level 20, left associativity).
Notation "s '⇒ t" := (tImp · s · t) (at level 51, right associativity).

Definition intf :=
  list type.

Reserved Notation " $\Gamma \vdash t : T$ " (at level 70, t at next level).

Reserved Notation " $\langle A \rangle$ ".

```

- a. Define `typing : intf → term → type → Prop` inductively. Note that `tFVar`, `tImp` and `tMP` are constructors for `tyForm`.
- b. Define a function `inj_form` using where `" $\langle A \rangle$ " := (inj_form A).` and prove:

```

Lemma type_inj_form  $\Gamma$  A :
   $\Gamma \vdash \langle A \rangle : \text{tyForm}$ .

```

- c. Define the reduction relation, the same as before, with extra rule

```

| red_MP A B :
  tMP ·  $\langle A \rangle$  ·  $\langle B \rangle$   $\succ$   $\langle \text{modus\_ponens } A B \rangle$ 

```

- d. Define semantic typing and prove the fundamental lemma.
- e. Define a function `tyinst : (nat → type) → type → type` and prove the following:

```

Definition implements ( $\Delta : \text{intf}$ )  $\sigma$   $\tau$  ( $\Gamma : \text{intf}$ ) :=
   $\forall n T,$ 
  nth_error  $\Gamma$  n = Some T  $\rightarrow$ 
   $\Delta \vdash \sigma n : \text{tyinst } \tau T$ .

```

```

Lemma instantiation  $\Delta$   $\sigma$   $\tau$   $\Gamma$  t T :
  implements  $\Delta$   $\sigma$   $\tau$   $\Gamma \rightarrow$ 
   $\Gamma \vdash t : T \rightarrow$ 
   $\Delta \vdash \text{inst } \sigma t : \text{tyinst } \tau T$ .

```

- f. Prove the following:

```

Definition tI : term :=
  tS · tK · tK.

```

```

Fixpoint lambda n s :=
  match s with
  | tVar m => if Nat.eqb n m then tI else tK · tVar m
  | tApp t u => tS · (lambda n t) · (lambda n u)
  | s => tK · s

```

`end.`

Definition `update` σ ($n : \text{nat}$) ($s : \text{term}$) $m := \text{if Nat.eqb } n \ m \ \text{then } s \ \text{else } \sigma \ m.$

Lemma `inst_lambda` σ n s $t :$
`inst` σ (`lambda` n s) $\cdot t \succ^* \text{inst} (\text{update } \sigma \ n \ t) \ s.$

g. We can implement K:

Definition `axK_impl` : `term` :=
`lambda` 0 (`lambda` 1 (`tVar` 0 $\Rightarrow$ `tVar` 1 $\Rightarrow$ `tVar` 0)).

Lemma `axK_impl_reduces` s $t :$
`axK_impl` $\cdot s \cdot t \succ^* s \Rightarrow t \Rightarrow s.$

h. Implement S as well.

5.3 Implementing LCF

a. Define `ILCF_impl` such that you can prove the following:

Definition `thm` : `type` := α 0.

Definition `ILCF` : `ctx` := [
 (* `mp` : *) `thm` $\Rightarrow$ `thm` $\Rightarrow$ `thm` ;
 (* `axK` : *) `tyForm` $\Rightarrow$ `tyForm` $\Rightarrow$ `thm` ;
 (* `axS` : *) `tyForm` $\Rightarrow$ `tyForm` $\Rightarrow$ `tyForm` $\Rightarrow$ `thm` ;
 (* `concl` : *) `thm` $\Rightarrow$ `tyForm`
].

Definition `ILCF_impl_ty` $n :=$
`match` n `with` 0 $\Rightarrow$ `tyForm` | $_ \Rightarrow \alpha \ n$ `end.`

Lemma `ILCF_implements` :
`implements` [] `ILCF_impl` `ILCF_impl_ty` `ILCF`.

b. We can now prove this implementation sound:

Lemma `LCF_sound` $t :$
`ILCF` $\vdash t : \text{thm} \rightarrow$
 [] $\vdash \text{inst } \text{ILCF_impl } t : \text{tyForm} \wedge$
`exists` $A, \text{inst } \text{ILCF_impl } t \succ^* \langle A \rangle \wedge [] \vdash A.$

c. And complete:

Lemma `LCF_complete` $A :$
 [] $\vdash A \rightarrow$
`exists` $t, \text{ILCF} \vdash t : \text{thm} \wedge \text{inst } \text{ILCF_impl } t \succ^* \langle A \rangle.$