Last time

Coq’s inductive types:
- Elimination rules and restrictions for inductives;
- Advanced examples of inductive definitions;
- Tactics for equality
- Paradoxes and the fixpoint operator;
- Tactics for case analysis and induction.
Plan for the second part

Today  More on Inductive Types and Sorts: Prop vs Type and extraction, well-founded recursion, proof techniques

31/1  Dependently-Typed Programming and Extraction: Subset Types, Inductive Families and Equality


14/2  Introduction to Homotopy Type Theory

7/3  Exam
Exercise/Project

To be sent to bruno.barras@inria.fr and matthieu.sozeau@inria.fr by March 7th 2022. INCLUDED.

THERE WILL BE NO DEADLINE EXTENSION!

Subject available on the course’s page.

Your submission should start with a short summary of your experience with the project indicating your previous experience with Coq or other proof assistants, the difficulties encountered and choices you made.

Questions are progressive and the whole subject requires only the lectures up to this one.

You don’t have to entirely complete it to get a good score.
Some Friendly Advice

- It is also an exercise in time management, your other exams are probably worth more than spending days on this :)
- Take a step back if you find yourself stuck in a proof, don’t throw yourself in endless proofs: the brute force video-game temptation can be strong.
- There are plenty of resources (Software Foundations, the reference manual, discourse forums, stack exchange, ... ) for technical issues. Do your research there!
- You can use your comrades advice of course, but try do it on your own, the goal is to build up your own expertise.
- It includes some programming, so testing your definitions before trying to prove anything about them is wildly encouraged and it will save you time.
A partial and short history of inductive types in DTT

- Martin-Löf Type Theory ('73), Automath (80’s)
- [Paulin-Mohring(1993)] Generic schema for inductive families in CIC. Reference paper for Coq (previous lecture and today)
- [Giménez(1996)] Coinductive definitions
- [Cornes(1997), McBride(1999)] Working with inductive families (lecture 6)
A partial and short history of inductive types in DTT

Above and beyond (not covered here):

- [Dybjer(2000)] Inductive-recursive definitions
- Inductive-inductive definitions (Dybjer, Setzer, ...),
- Ornaments (McBride, Dagand, ...),
- Dependent pattern-matching (McBride, Cockx, Sozeau, ...),
- Sized types (Abel, Sacchini...),
- Coinductive up-to techniques (Pous, Danielson, ...)
- Higher Inductive Types (Homotopy Type Theory)

In two words: exciting structures!
Today’s goals

- Play more with definitions of inductive types, programs and tactics on inductive values (match, fix, induction)
- Familiarize yourself with the sort system of Coq and the distinction between propositions and types.
Today

1. Introduction

2. Sorts
   - Hierarchy
   - Types
   - Propositions

3. Sorts and elimination rules
   - Sorts of inductive definitions
   - Elimination restrictions depending on sorts

4. Well-founded relations
   - Exercise: a variant of Markov’s principle
   - Accessibility
   - The Function tool

5. Induction and destruction smarts
Sorts in CIC

- The predicative Calculus of Inductive Constructions has sorts **Prop**, **Set** = **Type**₀, **Type**₁, **Type**₂, …
- **Prop** and **Set** are said small (because they do not type another sort)

  \[ \vdash \text{Prop, Set} : \text{Type}_1 \]

- sorts **Type**ᵢ (for \( i \geq 1 \)) are said large

  \[ \vdash \text{Type}_i : \text{Type}_{i+1} \]

In the standard mode (i.e. unless we turn on the -impredicative-set flag):

\[ \text{Prop} \subset \text{Set} = \text{Type}_0 \subset \text{Type}_1 \subset \text{Type}_2, \ldots \]
Predicative Types: the Type sorts

- **Type** is predicative:
  - Closed under products in the same universe:
    \[ A : \text{Type}_i, B : A \rightarrow \text{Type}_i \vdash \forall x : A, B x : \text{Type}_i \]
  - Quantification on a universe raises the level:
    \[ \vdash \text{Type}_i \rightarrow \text{Type}_i : \text{Type}_{i+1} \]
  
  This ensures consistency: \[ \vdash \text{Type}_i : \text{Type}_i \] is inconsistent (Girard’s paradox).

- **Type** is cumulative:
  \[ A : \text{Type}_i \vdash A : \text{Type}_{i+1} \]

- You don’t need to care about indices: typical ambiguity.
In Coq

lecture5_notes.v
Impredicative Propositions: the Prop sort

- Prop is impredicative: $\vdash \forall P : \text{Prop}, P : \text{Prop}$

  Subject to debate! Common to computer scientists: System $F^\omega$, at the core of Haskell.
Impredicative Propositions: the Prop sort

- **Prop** is **impredicative**: \( \vdash \forall P : \text{Prop}, P : \text{Prop} \)
  Subject to debate! Common to computer scientists: System F\(\omega\), at the core of Haskell.

- **Prop** evolved from earlier versions of Coq to represent proof-irrelevant propositions:

  \( \forall P : \text{Prop}, \forall x \ y : P, x = y \) (consistent **axiom**)

\[\]
Prop is impredicative: \( \vdash \forall P : \text{Prop}, P : \text{Prop} \)
Subject to debate! Common to computer scientists: System F\(\omega\), at the core of Haskell.

Prop evolved from earlier versions of Coq to represent proof-irrelevant propositions:

\[ \forall P : \text{Prop}, \forall x y : P, x = y \quad (\text{consistent axiom}) \]

Prop can be erased to recover the computational content of terms, this is known as extraction:
From Gallina:

\[
\text{fun } (x \ y : \text{nat}) (p : 0 < y) \Rightarrow \text{eqb } 0 (\text{rem } x \ y \ p)
\]

To ML:

\[
\text{fun } x \ y \Rightarrow \text{eqb } 0 (\text{rem } x \ y)
\]
Sorts and elimination rules
Conditions on sorts for the inductive definitions

- Arity and sort of the inductive definition
  \[ I : \forall (x_1 : A_1) \ldots (x_n : A_n)s \]

- A constructor has the form
  \[ c : \forall (y_1 : B_1) \ldots (y_p : B_p)I u_1 \ldots u_n \]

- Typing condition:
  \[ I : (x_1 : A_1) \ldots (x_n : A_n)s \vdash \forall (y_1 : B_1) \ldots (y_p : B_p)I u_1 \ldots u_n : s \]

- The sort of a predicative inductive definition (in the hierarchy \textbf{Type}) is the maximum of sorts of the types of the arguments of its constructors.

- Impredicative inductive definitions (type \textbf{Prop}) have no such constraint on arities (i.e. anything can be “injected” in a \textbf{Prop} inductive constructor, even large \textbf{Types}).
Restrictions of elimination depending on sorts

Elimination rule for type $\text{bool}$ (available for any possible sort $s$):

$$
\Gamma \vdash t : \text{bool} \quad \Gamma, x : \text{bool} \vdash A(x) : s \quad \Gamma \vdash t_1 : A(\text{true}) \quad \Gamma \vdash t_2 : A(\text{false})
$$

$$
\Gamma \vdash \text{match } t \text{ as } x \text{ return } A(x) \text{ with } \text{true } \Rightarrow t_1 \mid \text{false } \Rightarrow t_2 \text{ end} : A(t)
$$
Restrictions of elimination depending on sorts

Elimination rule for type \textit{bool} (available for any possible sort \(s\)):

\[ \Gamma \vdash t : \textit{bool} \quad \Gamma, x : \textit{bool} \vdash A(x) : s \quad \Gamma \vdash t_1 : A(\text{true}) \quad \Gamma \vdash t_2 : A(\text{false}) \]

\[ \Gamma \vdash (\text{match } t \text{ as } x \text{ return } A(x) \text{ with } \text{true } \Rightarrow t_1 \mid \text{false } \Rightarrow t_2 \text{ end}) : A(t) \]

Elimination rule for the type \textit{or A B} (only on \textit{Prop})

\[ \Gamma \vdash t : \textit{or A B} \quad \Gamma, p : A \vdash t_1 : C(\text{or_introl } p) \quad \Gamma, q : B \vdash t_2 : C(\text{or_intror } q) \]

\[ \Gamma \vdash \left( \text{match } t \text{ as } x \text{ return } C(x) \text{ with } \text{or_introl } p \Rightarrow t_1 \mid \text{or_intror } q \Rightarrow t_2 \text{ end} \right) : C(t) \]
### Terminology

- **Propositional** elimination: towards Prop sort only;
- **Weak** elimination: towards Prop and Set sorts only;
- **Strong** elimination: towards Type sort(s).
The elimination of inductive types in **Type** (predicative hierarchy) has no restriction:  
*weak & strong eliminations*

Elimination of inductive types in **Prop** is restricted:

- in general, one cannot build a type in **Type** by case on the proof-term in a proposition according to the implicit interpretation of **Prop** as **proof-irrelevant**:
  
  *propositional elimination only.*
The elimination of inductive types in **Type** (predicative hierarchy) has no restriction: *weak & strong eliminations*

Elimination of inductive types in **Prop** is restricted:

- in general, one cannot build a type in **Type** by case on the proof-term in a proposition according to the implicit interpretation of **Prop** as proof-irrelevant: *propositional elimination only.*
- exception **Singleton types**: if the type in **Prop** has zero constructor (absurdity) or a unique constructor whose arguments are in **Prop** (equality, conjunction . . .): *weak & strong eliminations.*
Rules on the sorts for the elimination

- The elimination of inductive types in **Type** (predicative hierarchy) has no restriction: *weak & strong eliminations*

- Elimination of inductive types in **Prop** is restricted:
  - in general, one cannot build a type in **Type** by case on the proof-term in a proposition according to the implicit interpretation of **Prop** as proof-irrelevant:
    *propositional elimination only.*
  - exception **Singleton types** : if the type in **Prop** has zero constructor (absurdity) or a unique constructor whose arguments are in **Prop** (equality, conjunction . . .):
    *weak & strong eliminations.*
  - exception **Impredicative set** : if the sort **Set** is impredicative, then a large inductive in **Set** can only be eliminated towards **Set**:
    *weak elimination.*
Impredicative Set

Fairly rare nowadays, allows definitions such as:

```
Inductive free_monad : Type -> Set :=
| unit {A : Type} : free_monad A
| bind {A B : Type} : free_monad A ->
  (A -> free_monad B) -> free_monad B
```

The constructors carry **Types** (A, B) themselves which can be ”larger” than **Set** and shouldn’t escape! The elimination restriction disallows for example:

```
match f in free_monad i return Type :=
| unit A => A
| bind A B m f => B
end
```

Otherwise we would have a variant of Hurkens paradox from a retraction between **Set** and **Type**.
In Coq

lecture5_notes.v
In practice in Coq

For each inductive definition of a type $I$, Coq defines automatically associated elimination schemes (when allowed):

- strong elimination (to $\text{Type}$) : $I_{\text{rect}}$
- elimination to small computational types (to $\text{Set}$) : $I_{\text{rec}}$
- elimination to logical propositions (to $\text{Prop}$) : $I_{\text{ind}}$

Moreover, by default these elimination schemes are:

- the dependent form when $I$ is computational (in sort $\text{Set}$ or $\text{Type}$);
- the non-dependent form when in $I$ is in sort $\text{Prop}$. 
Exercises

- Execute the command `Set Printing All`.
- Compare the types `True` and `unit`. What is the difference? Observe the consequence on the generated induction schemes.
- Prove the dependent scheme for `True`.
- Same questions with types `and` and `prod`.
- Same questions with types `sig` and `ex`.
- What is (each time) the main difference in behavior between these two datatypes?
- Execute the command `Unset Printing All`. 
Walkthrough: safe nth (lecture5_safe_nth.v)

Mixing Prop and Type

Example nth : forall A (l : list A),
{n : nat | n < length l} -> A.
Outline

1 Introduction

2 Sorts

3 Sorts and elimination rules

4 Well-founded relations
   - Exercise: a variant of Markov’s principle
     - A word on extraction
   - Accessibility
     - Another word on extraction
   - The Function tool

5 Induction and destruction smarts
The so-called Markov principle, or countable choice principle, states that for a boolean predicate on natural numbers,

$$ \neg\neg \exists n \ f(n) = \text{true} \iff \exists n \ f(n) = \text{true}, $$

In Coq’s logic, a variant of this principle states that the two notions of existence (\(\text{sig}\) and \(\text{ex}\)) coincide on boolean predicates when we turn \(\neg\neg \exists n \ f(n) = \text{true}\) into \(\text{ex}(\lambda n \Rightarrow f \ n = \text{true})\) and the second existential into a subset type \(\text{sig}(\lambda n \Rightarrow f \ n = \text{true})\). Recall \(\text{sig}\) is \(\{x : A \mid P \ x\}\) and \(\text{ex}\) is \(\text{exists} \ x : A. \ P \ x\).

State and prove the easy implication.

State the non-trivial implication. Why can’t it be proven directly?
Well-founded relations
Exercise: a variant of Markov’s principle

Markov’s principle II

- **Tactic needed for the rest:** intros destruct induction exists apply rewrite assumption contradiction

- **Require Import** Arith.

- **Open a Section** Markov, postulate a boolean predicate
  \( P : \text{nat} \rightarrow \text{bool} \) on natural numbers, and the hypothesis
  \( \text{exP} \) that the Prop existence of a witness for \( P \) holds.

- **Define the following predicate:**

  ```coq
  Inductive acc_nat (i : nat) : Prop :=
  \[\text{AccNat0} : P \ i \ = \ \text{true} \rightarrow \ acc\_nat \ i \]
  \[\text{AccNatS} : \ acc\_nat \ (\text{S} \ i) \rightarrow \ acc\_nat \ i.\]
  ```

- **What does it mean intuitively?**
Markov’s principle III (lecture5_markov.v)

- Prove:
  
  Lemma acc_nat_plus : forall x n : nat, 
  P (x + n) = true -> acc_nat n.

- Prove:
  
  Lemma acc_nat_0 : acc_nat 0.

- Prove:
  
  Lemma find_ex : forall n : nat, 
  acc_nat n -> {m : nat | P m = true}.

by induction on the hypothesis acc_nat n:
  
  - start with tactic fix 2 and observe the answer of Show Proof;
  - the tactic case_eq might be useful later on.
What happened?

Because of the restriction in elimination rules, this is not allowed:

```ocaml
Fixpoint find_ex_aux n (a : acc_nat n) : nat :=
  match a with
  | AccNat0 => ...
  | AccNatS a’ => if P n then n else find_ex_aux (S n) a’
end.
```

But this variant is accepted by the termination checker:

```ocaml
Fixpoint find_ex_aux n (a : acc_nat n) : nat :=
  if P n then n (* n is the witness *) else
  find_ex (S n) (match a with (* we go on searching *)
  | AccNat0 => ... (* cannot happen: P n = false in
                     this branch of the if *)
  | AccNatS a’ => find_ex_aux a’
end)
end.
```
A word on extraction

- Perform Extraction `find_ex_aux`, what’s left?
A word on extraction

- Perform Extraction find_ex_aux, what’s left?
- Extraction removes all proofs (understood as inhabitants of a type in Prop) [Letouzey(2004)].
- What to expect of extraction on acc_nat_0?
A word on extraction

- Perform Extraction `find_ex_aux`, what’s left?
- Extraction removes all proofs (understood as inhabitants of a type in `Prop`) [Letouzey(2004)].
- What to expect of extraction on `acc_nat_0`?
- What about extracting `find_ex`?
A word on extraction

- Perform Extraction find_ex_aux, what’s left?
- Extraction removes all proofs (understood as inhabitants of a type in Prop) [Letouzey(2004)].
- What to expect of extraction on acc_nat_0?
- What about extracting find_ex?
- Extraction also erases quantifications on types which are computationally-irrelevant.
- It works on closed terms only, can you see why?
Well-founded relations, constructively

**Inductive Acc** \((A : Type) (R : A \rightarrow A \rightarrow Prop) (x : A) : Prop :=

\(\text{Acc}_\text{intro} : (\forall y : A, R y x \rightarrow \text{Acc} R y) \rightarrow \text{Acc} R x\)
Exercises

Show the following lemma:

Section Accessibility.
Lemma not_Acc (a b : A) : R a b -> ¬ Acc R a -> ¬ Acc R b.

What is the definition of the constant well_founded?

What is well_founded_ind?

In the same section, show the following theorem:

Theorem not_decreasing (A : Set) (R : A -> A -> Prop) :
well_founded R ->
¬ (exists seq : nat -> A, 
(forall i : nat, R (seq (S i))(seq i))).

using the scheme well_founded_ind.

What about the reciprocal?
Exercises

- Define inductively the subterm relation on natural numbers
  \[
  \text{nat_subterm} : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}
  \]

- Show that it is well-founded.

- Using the definitions \text{clos_trans} and \text{wf_clos_trans} from Relations and Wellfounded show that the transitive closure of the subterm relation is wellfounded.

- Using the \text{Coq.Init.Wf.Fix} combinator, define \text{fact} : \text{nat} \rightarrow \text{nat} relying on well-founded induction on the subterm relation instead of the syntactic guard condition.

- Observe the \text{extraction} of \text{fact} using \text{Extraction} fact.
**Extraction**

Fix extracts to a general recursion combinator.

+ Acc proofs (which tend to be large) are no longer computed in the extracted code

! Using axioms, it is easy to mislead extraction and produce diverging code.

! Said otherwise the ML interfaces are not safe. There exists solutions to this (see Gradual Typing).
The Function toolbox helps the user encoding non-syntactically decreasing measures into this datatype.
The Function toolbox helps the user encoding non-syntactically decreasing measures into this datatype.

Definition slen (p : list nat * list nat) :=
    length (fst p) + length (snd p).
Function merge (p : list nat * list nat)
    {measure slen p} : list nat :=
    match p with
    | (nil, l2) => l2 | (l1, nil) => l1
    | ((x1 :: l1') as l1, (x2 :: l2') as l2) =>
        if leb x1 x2 then x1 :: merge (l1',l2)
        else x2 :: merge (l1,l2')
    end.

This generates obligations for the recursive calls.

- Observe the definition of merge.
- Test the function and try running it. Beware of the difference between Qed and Defined.
Outline

1. Introduction
2. Sorts
3. Sorts and elimination rules
4. Well-founded relations
5. Induction and destruction smarts
Inductive types naturally support case analysis. For inductive families/predicates, this generalizes to inversion/generation principles. E.g. suppose:

\[
\text{Inductive } \text{nat}_{\text{subterm}} : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop} := \\
| \text{nat}_{\text{S}}_{\text{subterm}} x : \text{nat}_{\text{subterm}} x (\text{S} x).
\]

\[
\text{Lemma } \text{nat}_{\text{subterm}}_{0}_{\text{empty}} x : \text{nat}_{\text{subterm}} x 0 \rightarrow \text{False}.
\]

Proof.
  intros H.
  inversion H.
Qed.

- There is only one introduction rule for \text{nat}_{\text{subterm}} and it cannot have a 0 as second argument.
- Inversion uses the injectivity and discrimination property of constructors to solve such goals.
Inversion II

Especially useful on complex inductive predicates! E.g. for a typing relation for STLC:

\[
\text{typing : context} \rightarrow \text{term} \rightarrow \text{Prop}
\]

It allows proving:

\[
\text{typing } \Gamma (\lambda x : \tau^1, t) (\tau^1 \rightarrow \tau^2) \Rightarrow \text{typing } (\Gamma, x : \tau^1) t \tau^2
\]

In one step as there is a single rule matching lambda’s in term position in the typing relation, and the domain of the lambda matches the domain of the arrow.
Stronger induction hypotheses

Important remarks:

- A key ingredient in proofs by induction is the possibility to strengthen the induction hypothesis;
- This might be a necessary ingredient in the proof;
- Because the statement of induction hypothesis is guessed by unification with the initial goal, this often boils down to strengthening the claim of the initial goal before applying the structural induction scheme;
- This can even allow to use on the fly seemingly alternate induction schemes, as illustrated in the next exercise.
The following induction scheme on natural numbers is useful:

\[\text{Lemma alt_nat_rec} (P : \text{nat} \rightarrow \text{Prop}) :\]
\[P \ 0 \rightarrow (\forall n, (\forall m, m \leq n \rightarrow P \ m) \rightarrow P \ (S \ n)) \rightarrow \forall n, P \ n.\]

Observe and complete the following proof script of this statement:

\[\text{Lemma alt_nat_rec} (P : \text{nat} \rightarrow \text{Prop}) :\]
\[P \ 0 \rightarrow (\forall n, (\forall m, m \leq n \rightarrow P \ m) \rightarrow P \ (S \ n)) \rightarrow \forall n, P \ n.\]

\[\text{Proof.}\]
\[\text{intros P0 Pind n.}\]
\[\text{generalize \ (le_refl \ n).}\]
\[\text{generalize n at 2.}\]
\[\text{intros m.}\]
\[\text{revert n.}\]
\[\text{induction \ m. todo}\]
Complete the following proof using the same reasoning pattern, without calling `alt_nat_rec` but generating on the fly in the script the appropriate induction hypothesis.

Section AlternateNatRecurrence.
Variable prime : nat -> Prop.
Variable div : nat -> nat -> Prop.
Hypothesis div_refl : forall n : nat, div n n.
Hypothesis prime2 : prime 2.
Hypothesis primeP : forall n : nat, 
  prime n \/ 
  exists d : nat, 2 <= d < n \/\ prime d \/\ div d n.

Lemma div_primeP (n : nat) : 2 <= n -> 
  exists d, div d n \/ prime d.
Proof. todo. Qed.
End AlternateNatRecurrence.
Exercises

Require Import Arith.
(* The `lia` tactic for linear arithmetic *)
Require Import Lia.

■ Show the following lemma:

Lemma NPeano_ltb_neg_geb (n m : nat) :
    NPeano.ltb n m = negb (NPeano.leb m n).

■ Define the following inductive relation:

Inductive leq_xor_gtn (m n : nat) :
    bool -> bool -> Set :=
    | LeqNotGtn : m <= n -> leq_xor_gtn m n true false
    | GtnNotLeq : n < m -> leq_xor_gtn m n false true.

■ Show the following lemma:

Lemma leqP (m n : nat) :
    leq_xor_gtn m n (NPeano.leb m n) (NPeano.ltb n m).
Exercises

Prove the following (dummy) result, observing what happens at the case analysis time:

Section SmartCaseAnalysis.

Variable T : Type.
Variables (t1 t2 : T).

Definition test1 (n m : nat) : T :=
    if (NPeano.leb n m) then t1 else t2.

Lemma test1P n m : n <= m -> test1 n m = t1.
...
Qed.

End SmartCaseAnalysis.
Bibliography

Cristina Cornes.
Conception d’un langage de haut niveau de représentation de preuves: Récurrence par filtrage de motifs, unification en présence de types inductifs primitifs, synthèse de lemmes d’inversion.

Peter Dybjer.
A general formulation of simultaneous inductive-recursive definitions in type theory.

Carlos Eduardo Giménez.
*Un calcul de constructions infinies et son application à la vérification de systèmes communicants.*

Pierre Letouzey.
*Programmation fonctionnelle certifiée: l’extraction de programmes dans l’assistant Coq.*
Thèse de doctorat, Université Paris-Sud, July 2004.

Conor McBride.
*Dependently Typed Functional Programs and Their Proofs.*

Christine Paulin-Mohring.