Last week and a bit of today ;) 

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.
A partial and short history of inductive types in DTT

- Martin-Lf Type Theory (’73), Automath (80’s)
- [Paulin, 93] Generic schema for inductive families in CIC. Reference paper for Coq (previous lecture and today)
- [Gimenez PhD] Coinductive definitions
- [Cornes, McBride] Working with inductive families
A partial and short history of inductive types in DTT

Above and beyond (not covered here):

- Inductive-recursive definitions (Dybjer)
- 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 (Propositions vs Types distinction)
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: Markov’s principle
   - Accessibility
   - The Function tool

5 Induction and destruction smarts
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)

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

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):

**Prop ⊂ Set = Type**$_0 ⊂ \text{Type}_1 ⊂ \text{Type}_2$, ⋯
Predicative Types: the Type sorts

- **Type** is predicative:
  - Closed under products in the same universe:
    \[ A : \text{Type}_i, B : A \to \text{Type}_i \vdash \forall x : A, B x : \text{Type}_i \]

- Quantification on a universe raises the level:
  \[ \vdash \text{Type}_i \to \text{Type}_i : \text{Type}_{i+1} \]

  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:

  \[ \forall P : \text{Prop}, \forall x y : P, x = y \quad \text{(consistent axiom)} \]
Prop is impredicative: \( \forall P : \text{Prop}, P : \text{Prop} \)
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
  \[ c : \forall (y_1 : B_1) \ldots (y_p : B_p) l \ u_1 \ldots u_n \]

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

- 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 \( bool \) (available for any possible sort \( s \)):

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

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

Elimination rule for type $bool$ (available for any possible sort $s$):

$$
\begin{align*}
\Gamma \vdash t : bool & \quad \Gamma, x : bool \vdash A(x) : s \\
\Gamma \vdash t_1 : A(true) & \quad \Gamma \vdash t_2 : A(false)
\end{align*}
$$

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

Elimination rule for the type $or A B$ (only on $Prop$)

$$
\begin{align*}
\Gamma \vdash t : or A B & \quad \Gamma, p : A \vdash t_1 : C(\text{or_introl } p) \\
\Gamma, x : or A B \vdash C(x) : Prop & \quad \Gamma, q : B \vdash t_2 : C(\text{or_intror } q)
\end{align*}
$$

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

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

Elimination of inductive types in \textbf{Prop} is restricted:

- In general, one cannot build a type in \textbf{Type} by case on the proof-term in a proposition according to the implicit interpretation of \textbf{Prop} as \textit{proof-irrelevant}:
  
  \textit{propositional elimination only}.

- Exception \textbf{Singleton types}: if the type in \textbf{Prop} has zero constructor (absurdity) or a unique constructor whose arguments are in \textbf{Prop} (equality, conjunction \ldots):
  
  \textit{weak & strong eliminations}. 
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.*
- partial exception: if the type in **Prop** has a unique constructor which arguments are either propositions of type **Prop** or small arities (type schemes which build in **Prop**), then elimination towards **Set** is allowed:  *weak elimination.*
In Coq

lecture4_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 Type) : $I_{\text{rect}}$
- elimination to small computational types (to Set) : $I_{\text{rec}}$
- elimination to logical propositions (to Prop) : $I_{\text{ind}}$

Moreover, by default these elimination schemes are:

- the dependent form when $I$ is computational (in sort Set or Type);
- the non-dependent form when in $I$ is in sort 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.
1 Introduction

2 Sorts

3 Sorts and elimination rules

4 Well-founded relations
   ■ Exercise: 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, the two notions of existence (\(\text{sig}\) and \(\text{ex}\)) coincide. Recall \(\text{sig}\) is \(\{x : A \mid P \ x\}\) and \(\text{ex}\) is \(\exists \ x : A. \ P \ x\).

State and prove the easy implication.

State the non-trivial implication. Why can’t it be proven directly?

Tactic needed for the rest:

\[\text{intros destruct induction exists apply rewrite assumption contradiction}\]
Markov’s principle II

- 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 $\text{Prop}$ existence of a witness for $P$ holds.
- Define the following predicate:

\[
\text{Inductive acc_nat (i : nat) : Prop :=} \\
\hspace{1cm} \text{AccNat0 : P i = true \rightarrow acc_nat i} \\
\hspace{1cm} \text{AccNatS : acc_nat (S i) \rightarrow acc_nat i.}
\]

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.
Exercise: Markov's principle

What happened?

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

```coq
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:

```coq
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 => ...
  | AccNatS a' => a'
end)
end.
```
A word on extraction

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

- Perform \texttt{Extraction find\_ex\_aux}, what’s left?
- Extraction removes all \textit{proofs} (understood as inhabitants of a type in \texttt{Prop}) [Letouzey’s PhD].
- What to expect of extraction on \texttt{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’s PhD].
- 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’s PhD].
- 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

\[
\text{Inductive } \text{Acc} \ (A : \text{Type}) \ (R : A \rightarrow A \rightarrow \text{Prop}) \ (x : A) : \text{Prop} := \\
\text{Acc_intro} : (\forall y : A, R y x \rightarrow \text{Acc} R y) \rightarrow \text{Acc} R x
\]

- \((\text{Acc} R x)\) **expresses in a concise way that:**
  - Either \(x\) is minimal (i.e. \(\forall y, \neg R y x\)): \text{Acc_intro} applies due to a contradiction
  - Or \(x\) is accessible from a minimal element in a finite (inductive) number of \(R\)-steps.

- It is a generalization of the \text{acc_nat} predicate we used in the exercise on the Markov principle.

- \(\forall x, \text{Acc} A R x\) expresses that \(R\) is a noetherian relation in \(A\) (every element is accessible).
Exercises

- Show the following lemma:

  Section Accessibility.
  Lemma not_Acc (a b : A) : R a b -> \not Acc R a -> \not 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 ->
  \not (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 \textit{extraction} of \text{fact} using \texttt{Extraction fact}. 
Extraction

\texttt{Fix} extracts to a \textit{general} recursion combinator.

+ \texttt{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 \textit{Function} toolbox helps the user encoding non-syntactically decreasing measures into this datatype.
Function (lecture5_function.v)

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.
1. Introduction
2. Sorts
3. Sorts and elimination rules
4. Well-founded relations
5. Induction and destruction smarts
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:

**Lemma** alt_nat_rec (P : nat → Prop) :

P 0 → (forall n, (forall m, m <= n → P m) → P (S n)) → forall n, P n.

Observe and complete the following proof script of this statement:

**Lemma** alt_nat_rec (P : nat → Prop) :

P 0 → (forall n, (forall m, m <= n → P m) → P (S n)) → forall n, P n.

**Proof.**

intros P0 Pind n.
generalize (le_refl n).
generalize n at 2.
intros m.
revert n.
induction m. todo
Stronger induction hypothesis

Complete the following proof using the same reasoning pattern, without calling \texttt{alt_nat_rec} but generating on the fly in the script the appropriate induction hypothesis.

Section AlternateNatRecurrence.
Variable prime : nat \to Prop.
Variable div : nat \to nat \to Prop.
Hypothesis div_refl : \forall n : nat, div n n.
Hypothesis prime2 : prime 2.
Hypothesis *primeP : \forall n : nat,
  prime n \nexists d : nat, 2 \leq d < n \land prime d \land div d n.

Lemma div_primeP (n : nat) : 2 \leq n \to
  \exists d, div d n \land prime d.
Proof. todo. Qed.
End AlternateNatRecurrence.
Exercises

Require Import Arith.
(* The 'omega' tactic for linear arithmetic *)
Require Import Omega.

- **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.