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, Equations

30/1 Meta-theory, axioms, models.


13/2 Meta-programming
Exercise/Project

There will be no deadline extension

- 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.
- You don’t have to entirely complete it to get a good score.
- Take a step back if you are stuck. If attempting a proof feels like being stuck in a video game – go for a walk!
- Read resources and ask questions on Zulip!
- It includes some programming, so testing your definitions first is 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)
Today’s goals

- The sort system of Coq and the distinction between propositions and types.
- Recursive functions which are not structurally recursive.
- Extract (correct) OCaml programs from Coq programs
1. Introduction

2. Sorts
   - Hierarchy
   - Types
   - Propositions

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

4. Extraction

5. Non-structural recursion, well-founded recursion
   - Exercise: $\Delta_1^0$ description
   - Accessibility
   - The Equations tool

6. Strengthened induction hypotheses and inversion
The predicative Calculus of Inductive Constructions has sorts \( \text{Prop}, \text{Set} = \text{Type}_0, \text{Type}_1, \text{Type}_2, \ldots \). \( \text{Prop} \) and \( \text{Set} \) are said small (because they do not type another sort)

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

Sorts \( \text{Type}_i \) (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 \texttt{-impredicative-set} flag):

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

- **Type** is predicative:
  - Closed under products in the same universe:
    \[ A : \text{Type}_i, B : A \to \text{Type}_i \vdash \forall x : A, B \ x : \text{Type}_i \]
  
- 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.
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 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)
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** can be erased to recover the computational content of terms, this is known as extraction:
  From GALLINA:
  ```latex
  \text{fun} \ (x \ y : \text{nat}) \ (p : 0 < y) \Rightarrow \text{eqb} \ 0 \ (\text{rem} \ x \ y \ p)
  ```
  To ML:
  ```latex
  \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 Type) is the maximum of sorts of the types of the arguments of its constructors.

- Impredicative inductive definitions (type Prop) have no such constraint on arities (i.e. anything can be “injected” in a Prop inductive constructor, even large Types).
Elimination rule for type $bool$ (available for any possible sort $s$):

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

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

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

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

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

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

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

- **Propositional** elimination: towards $\text{Prop}$ sort only;
- **Large** elimination: towards $\text{Type}$ sort(s).
The elimination of inductive types in \textbf{Type} (predicative hierarchy) has no restriction.

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}.
Rules on the sorts for the elimination

- The elimination of inductive types in **Type** (predicative hierarchy) has no restriction.

- 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.*
In Coq

$\text{lecture5_notes.v}$
For each inductive definition of a type $I$, Coq defines automatically associated elimination schemes (when allowed):

- strong elimination (to $Type$): $I_{\_rect}$
- elimination to small computational types (to $Set$): $I_{\_rec}$
- elimination to logical propositions (to $Prop$): $I_{\_ind}$

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 $Prop$. 
Impredicative encodings of logic

lecture5_notes.v
Exercises

- Define and verify the impredicative characterisations of or and ex.
- 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 Extraction

5 Non-structural recursion, well-founded recursion

6 Strengthened induction hypotheses and inversion
1 Introduction

2 Sorts

3 Sorts and elimination rules

4 Extraction

5 Non-structural recursion, well-founded recursion
   - Exercise: $\Delta^0_1$ description
   - Accessibility
   - The Equations tool

6 Strengthened induction hypotheses and inversion
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?
- Can we make it provable directly?
Non-structural recursion, well-founded recursion

Exercise: $\Delta^0_1$ description

In Coq

lecture5_notes.v
A word on extraction

- Perform extraction on before_witness_sig and G_sig.
- 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 on `before_witness_sig` and `G_sig`.
- 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 on `before_witness_sig` and `G_sig`.
- 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 sometimes erases quantifications on types which are `computationally-irrelevant`. 
Well-founded relations, constructively

**Inductive Acc** \((A : \text{Type}) \ (R : A \to A \to \text{Prop}) \ (x : A) : \text{Prop} := \)

\(\text{Acc\_intro} : (\forall y : A, R y x \Rightarrow \text{Acc} R y) \Rightarrow \text{Acc} R x\)
Exercises

- (Optional, skip for now) Exercise: Re-implement `safe_sig` returning just a natural number by directly implementing the program. Hint: Instead of \( f : \mathbb{N} \rightarrow \mathbb{B} \), use \( P : \mathbb{N} \rightarrow \text{Prop} \) with \( d : P n + \lnot P n \).

- Show the following lemma:

  ```adam
  Section Accessibility.
  Variables (A : Set) (R : A → A → Prop).
  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`?
Exercises

- In the same section, show the following theorem:

\[
\text{Theorem not_decreasing (A : Set)}
\]
\[
(R : A \to A \to \text{Prop}) :
\]
\[
\text{well_founded R} \to
\]
\[
\sim (\text{exists seq : nat} \to A,
\]
\[
\text{forall i : nat, R (seq (S i))(seq i))}.
\]

using the scheme well_founded_ind.

- What about the reciprocal?

- Define inductively the subterm relation on natural numbers

\[
\text{nat_subterm : nat} \to \text{nat} \to \text{Prop}
\]

- Show that it is well-founded.

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

- Using the Coq.Init.Wf.Fix combinator, define

\[
\text{fact : nat} \to \text{nat}
\]

relying on well-founded induction on the subterm relation instead of the syntactic guard condition.
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.
In Coq

lecture5_notes.v
1. Introduction

2. Sorts

3. Sorts and elimination rules

4. Extraction

5. Non-structural recursion, well-founded recursion

6. Strengthened induction hypotheses and inversion
Strengthened induction hypothesis

lecture5_notes.v
Inductive types naturally support case analysis. For inductive families/predicates, this generalizes to inversion/generation principles. E.g. suppose:

```coq
Inductive nat_subterm : nat -> nat -> Prop :=
  | nat_S_subterm x : nat_subterm x (S x).

Lemma nat_subterm_0_empty x : nat_subterm x 0 -> False.
Proof.
  intros H.
  inversion H.
Qed.
```

- There is only one introduction rule for `nat_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:

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

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 hypothesis

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
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 \texttt{AlternateNatRecurrence}.
Variable \texttt{prime} : nat \rightarrow Prop.
Variable \texttt{div} : nat \rightarrow nat \rightarrow Prop.
Hypothesis \texttt{div_refl} : \forall n : nat, \texttt{div} n n.
Hypothesis \texttt{prime2} : \texttt{prime} 2.
Hypothesis \texttt{primeP} : \forall n : nat,
    \texttt{prime} n \lor
    \exists d : nat, 2 \leq d < n \land \texttt{prime} d \land \texttt{div} d n.

Lemma \texttt{div_primeP} (n : nat) : 2 \leq n \rightarrow
    \exists d, \texttt{div} d n \land \texttt{prime} d.
Proof. \texttt{todo}. Qed.
End \texttt{AlternateNatRecurrence}.
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.
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.  