

# Coq workshop

LC2007

GClaramunt

# What is Coq?

Formal methods

- ↳ Programming logics

- ↳ Type theory

- ↳ Proof assistants

- ↳ Coq

# What is Coq?

- Formal language for program specification (type theory)
- Reasoning and proofs about the specification
- Mechanical assistant for proofs

# Proof assistant for Mathematicians and Logicians

# What I can do in Coq?

- Constructive logic
- Proof construction

$$\frac{\text{hypothesis}}{\text{goal}} \quad \left( \text{or} \quad \frac{\text{hypothesis}_1}{\text{goal}_1} \quad \frac{\text{hypothesis}_2}{\text{goal}_2} \quad \frac{\text{hypothesis}_n}{\text{goal}_n} \right)$$

A tactic transforms  $\frac{\Gamma(\text{hypothesis})}{\alpha(\text{goal})}$  into zero or more  $\frac{\Gamma_1}{\alpha_1} \quad \frac{\Gamma_2}{\alpha_2} \quad \dots \quad \frac{\Gamma_n}{\alpha_n}$

That's sufficient to build a proof of the original goal

# Propositional logic

Basic tactics

assumption (the goal is in the h )

$$\frac{\Gamma \quad H: \alpha}{\alpha}$$

# Propositional logic

Basic tactics

Introduction:

$$\frac{\Gamma}{\alpha \rightarrow \beta} \quad \text{intro H} \quad \frac{\Gamma \quad \text{H: } \alpha}{\beta}$$

# Propositional logic

## Basic tactics

"intro $\vee$ left"	$\frac{\Gamma}{\alpha \vee \beta}$	left	$\frac{\Gamma}{\alpha}$	
"intro $\vee$ right"	$\frac{\Gamma}{\alpha \vee \beta}$	right	$\frac{\Gamma}{\beta}$	
"intro $\wedge$ "	$\frac{\Gamma}{\alpha \wedge \beta}$	split	$\frac{\Gamma}{\alpha}$	$\frac{\Gamma}{\beta}$
" 'intro' $\_ \_$ "	$\frac{\Gamma}{\text{False}}$	absurd	$\frac{\Gamma}{\alpha}$	$\frac{\Gamma}{\sim \alpha}$



# Propositional logic

Basic tactics

Elimination

Apply

$$\frac{\Gamma \quad H: \alpha \rightarrow \beta}{\beta} \quad \text{apply H} \quad \frac{\Gamma \quad H: \alpha \rightarrow \beta}{\alpha}$$

In gral

$$\frac{\Gamma \quad H: \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \beta}{\beta} \quad \text{apply H} \quad \frac{\Gamma \quad H: \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \beta}{\alpha_1} \quad \frac{\Gamma \quad H: \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \beta}{\alpha_2} \quad \dots$$

# Propositional logic

## Basic tactics

### elimination

$$\begin{array}{ccc} \frac{\Gamma}{H: \alpha \vee \beta} & \text{elim H} & \frac{\Gamma}{H: \alpha \vee \beta} \quad \frac{\Gamma}{H: \alpha \vee \beta} \\ \hline g & & \frac{\alpha \rightarrow g}{\beta \rightarrow g} \\ \\ \frac{\Gamma}{H: \alpha \wedge \beta} & \text{elim H} & \frac{\Gamma}{H: \alpha \wedge \beta} \\ \hline g & & \frac{\alpha \rightarrow \beta \rightarrow g}{\alpha \rightarrow \beta \rightarrow g} \\ \\ \frac{\Gamma}{H: \text{False}} & \text{elim H} & \text{Proved!} \\ \hline g & & \end{array}$$

# Propositional logic

Other tactics

$$\frac{\Gamma}{\alpha} \quad \text{cut } \beta \quad \frac{\Gamma}{\beta \rightarrow \alpha} \quad \frac{\Gamma}{\beta}$$
$$\frac{\Gamma}{\alpha} \quad \text{exact } t \quad (\text{when } t \text{ is a proof of } \alpha)$$

# Propositional logic

Other tactics

Unfold ("expand definitions")

$$\frac{\Gamma}{\sim\alpha} \quad \text{unfold not} \quad \frac{\Gamma}{\alpha \rightarrow \text{False}}$$

$$\frac{\Gamma}{\beta \leftrightarrow \alpha} \quad \text{unfold iff} \quad \frac{\Gamma}{(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)}$$

in hypothesis: unfold *\*name\** in *\*h\**

# Propositional logic

Other

(classical logic)  $a \vee \sim a$

(constructive tautologies)  $\text{auto}$

Proof time!

# Higher order predicate calculus

# Higher order predicate calculus

forall  $x:U, a$

exists  $x:U, a$

e.g.

forall  $f\ g: \text{nat} \rightarrow \text{nat}, (\text{forall } x:\text{nat}, f\ x = g\ x) \rightarrow f=g$

forall  $k:U, A(k) \rightarrow (\text{exists } x:U, A(x))$



# Higher order predicate calculus - Tactics

forall

Intro

$$\frac{\Gamma}{\text{forall } x:A, B}$$

intro x

$$\frac{\Gamma \quad x:A}{B}$$

# Higher order predicate calculus - Tactics

forall

Elim

$$\frac{\Gamma \quad H:\text{forall } x:A, Y}{B}$$

apply H - Proved!

(if exists  $a:A$  /  $B = Y[x:=a]$  )

$$\frac{\Gamma \quad H:\text{forall } (x_1:A_1)(x_2:A_2)\dots, Y}{B}$$

apply H or apply H (a1 a2 ...) - Proved!

(if exists  $a_1:A_1, a_2:A_2[x_1:=a_1]\dots$  /  $B=Y[x_1:=a_1,\dots,x_n:=a_n]$ )

$$\frac{\Gamma \quad H:\text{forall } x_1:A_1, A_2 \rightarrow \text{forall } x_3:A_3, Y}{B}$$

apply H

$$\frac{\Gamma \quad H:\text{forall } x_1:A_1, A_2 \rightarrow \text{forall } x_3:A_3, Y}{A_2[x_1:=a_1]}$$

(if exists  $a_1:A_1$  and  $a_3:A_3[x_1:=a_1]$  /  $B= Y[x_1:=a_1, x_3:=a_3]$ )

# Higher order predicate calculus - Tactics

exists

Intro

$$\frac{\Gamma}{\text{exists } x:U, A(x)} \quad \text{exists } t \quad \frac{\Gamma}{A(t)} \quad (\text{where } t:U \text{ is a witness})$$

Elim

$$\frac{\Gamma}{\text{H:exists } x:U, A(x)} \quad \text{elim H} \quad \frac{\Gamma}{\text{H:exists } x:U, A(x)} \quad \frac{}{\text{forall } y:U, A(y) \rightarrow B} \quad (\text{where } y \text{ is a new variable not in } B)$$

# Higher order predicate calculus - Tactics

Other tactics

reflexivity, symmetry, transitivity

rewrite -> \*term\*

replace a with b

Show Proof

Show Tree

Proof time!

# Proof assistant for Programmers

# The Curry-Howard correspondence

- direct relationship between computer programs and mathematical proofs
- generalization of a syntactic analogy between systems of formal logic and computational calculi

“The Curry–Howard correspondence is the observation that two families of seemingly unrelated formalisms—namely, the proof systems on one hand, and the models of computation on the other—are in fact the same kind of mathematical objects.”

# Coq programming language

Typed lambda calculus with

higher order

parametric functions

dependent types

CC

(calculus of constructions)

inductive types

CCI

(calculus of constructions with inductive types)

coinductive types

$\text{CCI}^\infty$

(calculus of constructions with inductive  
and coinductive types)



# Programing proofs

H1: (x:A) B -> C

H2: B

z:A

=====

C

exact ((H1 z) H2 )

# Syntax

```
T := Set | Type | Prop
| x
| c
| (T T)
| [ x : T ] T
| ( x : T ) T
| T -> T
| Inductive x [x:T,..x:T] : T := c:T | ... | c:T
| <T> match T with T=>T | ... | T => T end
| Fixpoint x [x:T,..x:T] : T := T
```

variables  
defined constants  
application  
abstraction  
product  
function type  
inductive def.  
case analysis  
recursive def.

# Inductive types (Set)

```
Inductive nat : Set :=  
  0 : nat  
  | S : nat -> nat
```

```
Inductive bool : Set :=  
  true : bool  
  | false : bool
```

```
Inductive natlist : Set :=  
  nil : natlist  
  | cons : nat -> natlist -> natlist
```

# Parametric inductive types

```
Inductive list (A:Set) : Set :=  
  nil : list A  
| cons : A -> list A -> list A
```

# Inductive type families

```
Inductive array(A:Set) : nat -> Set :=  
  empty : array A 0  
| add : forall n:nat, A -> array A n -> array A (S n)
```

# Mutually inductive type families

Inductive

```
tree (A:Set) : Set :=  
  node: A -> forest A -> tree A
```

with

```
forest (A:Set) : Set :=  
  empty_f : forest A  
  | add_tree: tree A -> forest A -> forest A
```

# Inductive predicates

## Inductive predicates

```
Inductive Even : nat -> Prop :=  
  e0 : Even 0  
  | eSS : forall n:nat, Even n -> Even (S (S n))
```

# Inductive definitions - Consequences

Case analysis (“pattern matching”) - Functions

Recursion

Case analysis - Propositions

Induction

Destructors



# Functions

(case analysis)

```
Definition pred :=  
  fun n:nat => match n with  
    0 => 0  
    | S m => m  
  end.
```

# Functions

- Constructors application:  
    `apply c i`  
    `constructor i (= intros ; apply c i) / constructor`
- Constructors discrimination:  
    `discriminate H` (proves anything if  $H: t_1 = t_2$  , with  $t_1$  and  $t_2$  created with different constructors)
- Constructors injectivity:  
    `injection H` (removes same constructors from an equality)
- In general...  
    `simplify_eq` (applies `discriminate` or `injection`)

# Recursive functions

## Fixpoint

```
Fixpoint f (x 1 : I 1 ) ... (x n :I n ) : Q := e
```

```
Fixpoint add (n m:nat) {struct n}: nat :=
```

```
match n with
```

```
  0    => m
```

```
  | S k => S (add k m)
```

```
end.
```

```
Fixpoint even (n:nat) : bool :=
```

```
  match n with 0 => true  | S k => odd k      end
```

```
with odd (n:nat) : bool :=
```

```
  match n with 0 => false | S k => even k      end.
```

# Case analysis - Propositions

**To prove a property  $P$  ( $: \text{Prop}$ ) by cases of an object  $x$  of a inductive type  $I$**

Tactics:

- `case x`: generates cases according the definition of  $I$ .
- `destruct x`: applies intros and then case

# Induction

## Induction

To prove a property  $P$  using the primitive induction principle associated to an inductive definition of a type  $I$

Tactics:

- `elim x`:
  - generates cases according the definition of  $x$ , with their inductive hypothesis
- `induction x`:
  - applies intros and then `elim`

`elim` = `apply <destructor>`

# Destructors

When an inductive type `I` is defined, Coq generates three constants corresponding to the recursion and induction principles:

- `I_ind` is the induction principle for `Prop`
  - Implements the structural induction principle for objects of `I`
- `I_rec` is the induction principle for `Set`
  - allows recursive definitions over objects of `I`
- `I_rect` is the induction principle for `Type`
  - permite definir familias recursivas de tipos

Let's code!

# Program extraction

Extract correct code to:

Scheme

Haskell

OCaml

( but also F#, Rust, Scala )



# References

<http://coq.inria.fr/>

<https://coq.inria.fr/tutorial-nahas>

<https://coq.inria.fr/tutorial/>

<https://x80.org/collacoq/>

<https://www.cis.upenn.edu/~bcpierce/sf/current/index.html>

Thank you!