Coq

LASER 2011 Summerschool
Elba Island, Italy

Christine Paulin-Mohring

`http://www.lri.fr/~paulin/LASER`

Université Paris Sud & INRIA Saclay - Île-de-France

September 2011

Lecture 2 : Using simple inductive definitions

- Covers section 3 (+ part of 4) of course notes
- Example : `board.v`
- Challenge 5.4 (Needham-Schroeder protocol)

Basic objects

- sorts: $\mathrm{Prop} : \mathrm{Type}_1$, $\mathrm{Type}_i : \mathrm{Type}_{i+1}$
- variables
- one product: $\forall x : A, B$ (includes $A \to B$)
- function abstraction: $\mathbf{fun}\ x : A \Rightarrow t$
- function application: $t\ u$

Computation:

- $(\mathbf{fun}\ x : A \Rightarrow t)\ u \equiv t[x \leftarrow u]$

Basic rules

Environment ($\Gamma\vdash$) :

$$\frac{}{[]\vdash} \qquad \frac{\Gamma\vdash A : s}{\Gamma, x : A\ \vdash}$$

Constant and variable:

$$\frac{\Gamma\ \vdash}{\Gamma\vdash s_1 : s_2} \qquad \frac{\Gamma\ \vdash \quad (x,A)\in\Gamma}{\Gamma\vdash x : A}$$

Product:

$$\frac{\Gamma\vdash A : s_1 \quad \Gamma, x : A\vdash B : s_2}{\Gamma\vdash \forall x : A, B : s_2}$$

Function:

$$\frac{\Gamma, x : A\vdash t : B}{\Gamma\vdash \textbf{fun}\ x : A\Rightarrow t : \forall x : A, B} \qquad \frac{\Gamma\vdash t : \forall x : A, B \quad \Gamma\vdash u : A}{\Gamma\vdash t\,u : B[x\leftarrow u]}$$

Computation:

$$\frac{\Gamma\vdash t : A \quad \Gamma\vdash B : s \quad A\equiv B}{\Gamma\vdash t : B}$$

Local definitions

let $x := e$ **in** t

Inductive definitions

See next lecture

Modules

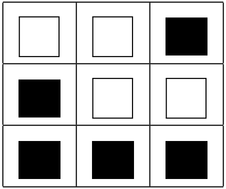
independant level

Outline

- Introduction
- Basics of Coq system
- Using simple inductive definitions
  - The board example
  - Properties of inductive definitions
- Functional programming with Coq
- Automating proofs

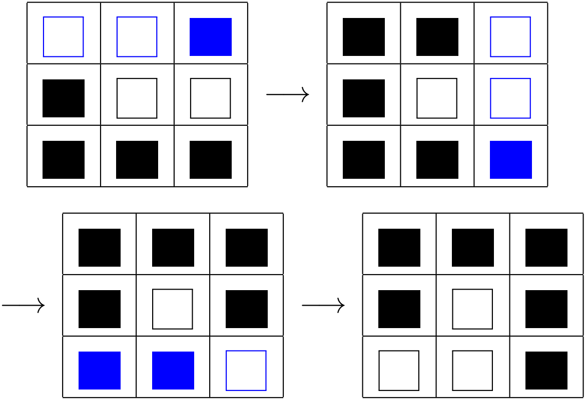
The board example

Nine bicolor tokens (one face black and one face white) are placed on a 3×3 board.



- turn at each step either one line or one column (it alternates the color of the tokens)
- study when a configuration is reachable from a starting position.

Example



A little bit of programming

- Booleans to represent colors (`true` is white)
- Triple of colors to represent lines
- Triple of lines to represent boards

```
Coq? Definition color := bool.

Coq? Definition line : Type := (color * color * color)%type.

Coq? Definition board : Type := (line * line * line)%type.

Coq? Definition inv_line (l:line) : line :=
Coq?   match l with (x,y,z) => (negb x,negb y,negb z) end.
```

How to define the operation which turns a line ?
Which line ?

turning the tokens on a line

We index the line by an integer : 0, 1 or 2

```
Coq? Definition turn_line (i:nat) (b:board) :=
Coq?   match b with (l0,l1,l2) =>
Coq?      match i with 0 => (inv_line l0,l1,l2)
Coq?                 | 1 => (l0,inv_line l1,l2)
Coq?                 | 2 => (l0,l1,inv_line l2)
Coq?      end
Coq?   end.
Error: Non exhaustive pattern-matching: no clause found for pattern
S (S (S _))
```

Making the function total

```
Coq? Definition turn_line (i:nat) (b:board) :=
Coq?   match b with (l0,l1,l2) =>
Coq?      match i with 0 => (inv_line l0,l1,l2)
Coq?                 | 1 => (l0,inv_line l1,l2)
Coq?                 | 2 => (l0,l1,inv_line l2)
Coq?                 | _ => (l0,l1,l2)
Coq?      end
Coq? end.
turn_line is defined
```

Turning a column

```
Coq? Definition inv_elt (i:nat) (l:line) : line :=
Coq?   match l with (x,y,z) =>
Coq?       match i with 0 => (negb x,y,z)
Coq?                  | 1 => (x,negb y,z)
Coq?                  | 2 => (x,y,negb z)
Coq?                  | _ => (x,y,z)
Coq?       end
Coq? end.

Coq? Definition turn_col (i:nat) (b:board) :=
Coq?   match b with (l0,l1,l2) =>
Coq?          (inv_elt i l0, inv_elt i l1, inv_elt i l2)
Coq?   end.
```

Similar code, generalisation ?

Another approach

Work with polymorphic triples

```
Coq? Section Polymorphic_Triple.

Coq?   Variable M : Type.
M is assumed

Coq?   Inductive triple : Type := Triple : M -> M -> M -> triple.
triple is defined
triple_rect is defined
triple_ind is defined
triple_rec is defined

Coq?   Definition triple_x (m : M) : triple := Triple m m m.
triple_x is defined
```

Operators on triples

```
Coq? Variable f : M -> M.

Coq? Definition triple_map (t: triple) : triple :=
Coq?   match t with (Triple a b c) => (Triple (f a) (f b) (f c))
Coq?   end.

Coq? Inductive pos : Type := A : pos | B : pos | C : pos.

Coq? Definition triple_map_select (p : pos) (t:triple) : triple :=
Coq?   match t with (Triple a b c) =>
Coq?     match p with A => (Triple (f a) b c)
Coq?                | B => (Triple a (f b) c)
Coq?                | C => (Triple a b (f c))
Coq?     end
Coq?   end.
```

Indexing the elements

```
Coq? Definition proj_triple (p: pos) (t: triple) : M :=
Coq?   match t with (Triple a b c) =>
Coq?     match p with A => a | B => b | C => c end
Coq?   end.
proj_triple is defined

Coq? Check triple_map.
triple_map
     : triple -> triple

Coq? End Polymorphic_Triple.

Coq? Check proj_triple.
proj_triple
     : forall M : Type, pos -> triple M -> M

Coq? Check triple_map.
triple_map
     : forall M : Type, (M -> M) -> triple M -> triple M
```

Modeling the board

```
Coq? Inductive color : Type := White : color | Black : color.

Coq? Definition turn_color (c: color) : color :=
Coq?   match c with White => Black | Black => White end.

Coq? Definition board := triple (triple color).

Coq? Definition turn_row (p: pos) : board -> board :=
Coq?   triple_map_select (triple_map turn_color) p.

Coq? Definition turn_col (p: pos) : board -> board :=
Coq?   triple_map (triple_map_select turn_color p).

Coq? Definition proj_board (x y: pos) (b:board) : color :=
Coq?  proj_triple y (proj_triple x b).
```

Computing inside Coq

```
Coq? Eval compute in (turn_color White).
     = Black
     : color

Coq? Definition white_board : board
Coq?    := triple_x (triple_x White).
white_board is defined

Coq? Notation "[ x | y | z ]" := (Triple x y z).
Setting notation at level 0.

Coq? Eval compute in (turn_row A (turn_col B white_board)).
     = [[Black | White | Black] | [White | Black | White]
       | [White | Black | White]]
     : board
```

Outline

- Introduction
- Basics of Coq system
- Using simple inductive definitions
  - The board example
  - Properties of inductive definitions
- Functional programming with Coq
- Automating proofs

Rules for (mutual)-inductive definitions

$$\frac{\Gamma \vdash \quad \textbf{Ind}(I_1 : A_1, \ldots, I_k : A_k)(c_1 : C_1 | \ldots | c_n : C_n) \ \texttt{well-formed}}{\Gamma, \textbf{Ind}(I_1 : A_1, \ldots, I_k : A_k)(c_1 : C_1 | \ldots | c_n : C_n) \vdash}$$

Now we assume $\textbf{Ind}(I_1 : A_1, \ldots, I_k : A_k)(c_1 : C_1 | \ldots | c_n : C_n) \in \Gamma$.

New objects available for types and constructors:

$$\frac{}{\Gamma \vdash I_i : A_i} \qquad \frac{}{\Gamma \vdash c_i : C_i}$$

Examples

```
type bool = true | false.
type nat = O | S of nat.

Coq? Print bool.
Inductive bool : Set :=  true : bool | false : bool

Coq? Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
For S: Argument scope is [nat_scope]
```

Logical definitions:

```
Coq? Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

Infinite branching:

```
Coq? Inductive ord : O : ord | S : ord -> ord
Coq?                 | lim : (nat -> ord) -> ord.
```

Examples

```
type bool = true | false.
type nat = O | S of nat.

Coq? Print bool.
Inductive bool : Set :=  true : bool | false : bool

Coq? Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
For S: Argument scope is [nat_scope]
```

Logical definitions:

```
Coq? Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

Infinite branching:

```
Coq? Inductive ord : O : ord | S : ord -> ord
Coq?                 | lim : (nat -> ord) -> ord.
```

Examples

```
type bool = true | false.
type nat = O | S of nat.

Coq? Print bool.
Inductive bool : Set :=  true : bool | false : bool

Coq? Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
For S: Argument scope is [nat_scope]
```

Logical definitions:

```
Coq? Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

Infinite branching:

```
Coq? Inductive ord : O : ord | S : ord -> ord
Coq?                 | lim : (nat -> ord) -> ord.
```

Inductive predicates

$$\frac{}{n \leq n} \qquad \frac{n \leq m}{n \leq S\, m}$$

```
Coq? Print le.
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n | le_S : forall m : nat, n <= m -> n <= S m
```

Indexed types:

```
Coq? Inductive vect (A:Type) : nat -> Type :=
Coq?   v0 : vect 0
Coq? | v1 : forall n, A -> vect n -> vect (S n).
```

Inductive predicates

$$\frac{}{n \leq n} \qquad \frac{n \leq m}{n \leq S\,m}$$

```
Coq? Print le.
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n | le_S : forall m : nat, n <= m -> n <= S m
```

Indexed types:

```
Coq? Inductive vect (A:Type) : nat -> Type :=
Coq?   v0 : vect 0
Coq? | v1 : forall n, A -> vect n -> vect (S n).
```

Inductive predicates

$$\frac{}{n \leq n} \qquad \frac{n \leq m}{n \leq S\,m}$$

```
Coq? Print le.
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n | le_S : forall m : nat, n <= m -> n <= S m
```

Indexed types:

```
Coq? Inductive vect (A:Type) : nat -> Type :=
Coq?   v0 : vect 0
Coq? | v1 : forall n, A -> vect n -> vect (S n).
```

Elimination rule for inductive types

- Pattern-matching construction (+ possibly fixpoint)
- Dependent typing

$$
\begin{array}{l}
\textbf{match } c \textbf{ as } x \textbf{ in } I\ \vec{y}\ \ \textbf{returns}\ \ P(\vec{y}, x) \\
\quad \textbf{with} \begin{array}{l} \mid c_{i_1}\ \vec{x_1} \Rightarrow f_1 \\ \mid \ldots \\ \mid c_{i_p}\ \vec{x_p} \Rightarrow f_p \end{array} \\
\textbf{end}
\end{array}
$$

- has type $P(\vec{t}, c)$ when c has type $I\ \vec{t}\ c$
- reduction as expected when $c = c_{i_k}\ \vec{u}$:

$$f_k[\vec{x_k} \leftarrow \vec{u}]$$

Elimination rule for inductive types

- Pattern-matching construction (+ possibly fixpoint)
- Dependent typing

$$\begin{array}{l} \textbf{match } c \\ \quad \textbf{with } \mid c_{i_1}\ \vec{x_1} \Rightarrow f_1 \\ \qquad\quad\ \mid \dots \\ \qquad\quad\ \mid c_{i_p}\ \vec{x_p} \Rightarrow f_p \\ \textbf{end} \end{array}$$

- has type $P(\vec{t}, c)$ when c has type $I\,\vec{t}\,c$
- reduction as expected when $c = c_{i_k}\ \vec{u}$:

$$f_k[\vec{x_k} \leftarrow \vec{u}]$$

Elimination rule for inductive types

- Pattern-matching construction (+ possibly fixpoint)
- Dependent typing

$$\begin{array}{l} \textbf{match } c \\ \quad \textbf{with} \mid c_{i_1}\ \vec{x_1} \Rightarrow f_1 \\ \qquad\qquad \mid \ldots \\ \qquad\qquad \mid c_{i_p}\ \vec{x_p} \Rightarrow f_p \\ \textbf{end} \end{array}$$

- has type $P(\vec{t}, c)$ when c has type $I\,\vec{t}\,c$
- reduction as expected when $c = c_{i_k}\ \vec{u}$:

$$f_k[\vec{x_k} \leftarrow \vec{u}]$$

Elimination rule for inductive types

- Pattern-matching construction (+ possibly fixpoint)
- Dependent typing

$$\begin{array}{ll} \textbf{match } c & \\ \quad \textbf{with} & \mid c_{i_1}\ \vec{x_1} \Rightarrow f_1 \\ & \mid \ldots \\ & \mid c_{i_p}\ \vec{x_p} \Rightarrow f_p \\ \textbf{end} & \end{array}$$

- has type $P(\vec{t}, c)$ when c has type $I\,\vec{t}\,c$
- reduction as expected when $c = c_{i_k}\ \vec{u}$:

$$f_k[\vec{x_k} \leftarrow \vec{u}]$$

Examples

```
Coq? Check color_ind.
color_ind : forall P : color -> Prop,
   P White -> P Black -> forall c : color, P c

Coq? Check triple_ind.
triple_ind: forall (M : Set) (P : triple M -> Prop),
       (forall m m0 m1 : M, P [m | m0 | m1])
       -> forall t : triple M, P t
```

Dependent case analysis

```
Coq? Definition color_case (P : Type) (fw fb: P) (c:color) : P
 := match c  with | White => fw | Black => fb end

Coq? Print color_rect.
color_rect =
fun (P : color -> Type) (f : P White) (f0 : P Black) (c : color) =>
match c as c0 return (P c0) with
| White => f
| Black => f0
end
: forall P : color -> Type, P White -> P Black
  -> forall c : color, P c
```

Corresponding tactics

$$\frac{\Gamma \vdash h : I\,\vec{t}\,c \qquad (\Gamma \vdash {\color{blue}?} : \forall \vec{z}, C[\vec{y} \leftarrow \vec{u}, x \leftarrow c_i\,\vec{z}])_i}{\Gamma \vdash {\color{blue}?} : C[\vec{y} \leftarrow \vec{t}, x \leftarrow h]}$$

- `case` h
- `destruct` h : composed tactic doing substitution in context, intros
- complex rule : abstraction of C (multiple choices),
- generalisation when I appears in the conclusion of h: find instances and generates extra subgoals.

Equality (intensional)

`reflexivity`	$\dfrac{t \equiv u}{\Gamma \vdash ? : t = u}$
`rewrite` h	$\dfrac{\Gamma \vdash h : t = u \quad \Gamma \vdash ? : C[x \leftarrow u]}{\Gamma \vdash ? : C[x \leftarrow t]}$
`symmetry`	$\dfrac{\Gamma \vdash ? : u = t}{\Gamma \vdash ? : t = u}$
`transitivity` v	$\dfrac{\Gamma \vdash ? : t = v \quad \Gamma \vdash ? : v = u}{\Gamma \vdash ? : t = u}$
`f_equal`	$\dfrac{\Gamma \vdash ? : f = g \quad (\Gamma \vdash ? : t_i = u_i)_i}{f\, t_1 \ldots t_n = g\, u_1 \ldots u_n}$

Equality and inductive definitions

- Different constructors give different values
- Constructors are injective
- Derived from the elimination rule

Non confusion

```
Coq? Definition is_white : color -> Prop :=
Coq?   fun c => match c with White => True | Black => False end.

Coq? Lemma black_not_white : ~ (Black = White).

Coq? intro H.
1 subgoal

  H : Black = White
  ============================
   False

Coq? change (is_white Black).
1 subgoal

  H : Black = White
  ============================
   is_white Black

Coq? rewrite H; simpl.
1 subgoal

  H : Black = White
  ============================
   True
```

Tactic `discriminate`

```
Coq? Restart.
1 subgoal

  ============================
   Black <> White

Coq? discriminate.
Proof completed.

Coq? Qed.
discriminate.
black_not_white is defined
```

Extension : `discriminate` h
with h a proof of equality : $C[c_i] = C[c_j]$

Injectivity

```
Coq? Lemma inj1 : forall M (x1 x2 x3 y1 y2 y3:M),
Coq?     [x1 | x2 | x3] = [y1 | y2 | y3] -> x1 = y1.
```

Projection:

```
Coq? Definition pr1 M (t:triple M) : M
Coq?   := let (x,y,z) := t in x.

Coq? intros; change (pr1 [x1 | x2 | x3] = pr1 [y1 | y2 | y3]).
1 subgoal

  M : Type
  x1 : M
  x2 : M
  x3 : M
  y1 : M
  y2 : M
  y3 : M
  H : [x1 | x2 | x3] = [y1 | y2 | y3]
  ============================
   pr1 [x1 | x2 | x3] = pr1 [y1 | y2 | y3]

Coq? rewrite H; trivial.
Proof completed.
```

injectivity

```
Coq? Restart.

Coq? intros.

Coq? injection H.
1 subgoal

  M : Type
  x1 : M
  x2 : M
  x3 : M
  y1 : M
  y2 : M
  y3 : M
  H : [x1 | x2 | x3] = [y1 | y2 | y3]
  ============================
   x3 = y3 -> x2 = y2 -> x1 = y1 -> x1 = y1
```

tactic `subst` x finds an hypothesis $x = t$ and replaces x by t.

Accessibility

move b_1 b_2 if b_2 is obtained from b_1 with one valid move.

```
Coq? Inductive move (b1:board) : board -> Prop :=
Coq?   move_row : forall p, move b1 (turn_row p b1)
Coq? | move_col  : forall p, move b1 (turn_col p b1).
```

Reflexive-transitive closure

```
Coq? Inductive moves (b1:board): board -> Prop :=
Coq?   moves_init : moves b1 b1
Coq? | moves_step : forall (b2 b3:board),
Coq?    moves b1 b2 -> move b2 b3 -> moves b1 b3.

Coq? Hint Constructors move moves.
```

Properties of moves

```
Coq? Lemma move_moves
Coq?  : forall (b1 b2:board), move b1 b2 -> moves b1 b2.

Coq? intros; apply moves_step with b1; trivial.

Coq? Save.

Coq? Lemma reachable : moves start target.

Coq? apply moves_step with (turn_row A start); auto.

Coq? replace target with (turn_row B (turn_row A start)); auto.

Coq? Save.
```

Induction on predicate

```
Coq? Lemma moves_trans
Coq?   : forall (b1 b2 b3:board),
Coq?     moves b1 b2 -> moves b2 b3 -> moves b1 b3.
```

Induction principle capture minimality

```
Coq? Check moves_ind.
moves_ind : forall (b1 : board) (P : board -> Prop),
  P b1 ->
  (forall b2 b3 : board, moves b1 b2 -> P b2 -> move b2 b3 -> P b3)
  -> forall b : board, moves b1 b -> P b

Coq? induction 2.
2 subgoals

  b1 : board
  b2 : board
  H : moves b1 b2
  ============================
   moves b1 b2
subgoal 2 is:
 moves b1 b3
```

Proving a state is not reachable

- A white board cannot be accessed from start
- Assuming `moves start white_board`, derive a contradiction
- Find an appropriate invariant
- any idea ?
- parity of white tokens at the 4 corners

Proving a state is not reachable

- A white board cannot be accessed from start
- Assuming `moves start white_board`, derive a contradiction
- Find an appropriate invariant
- any idea ?
- parity of white tokens at the 4 corners

Proving a state is not reachable

- A white board cannot be accessed from start
- Assuming `moves start white_board`, derive a contradiction
- Find an appropriate invariant
- any idea ?
- parity of white tokens at the 4 corners

Proving a state is not reachable

- A white board cannot be accessed from start
- Assuming `moves start white_board`, derive a contradiction
- Find an appropriate invariant
- any idea ?
- parity of white tokens at the 4 corners

Summary

What you should have learned sofar

- Load and query libraries
- Do basic logical reasoning
- Model simple problems

From the theoretical point of view:

- Every type (proposition) is a sort, a product or an inductive.
- Basic term (proof) construction is a variable, a constructor, a function, an application or a **match**.