Coq

LASER 2011 Summerschool
Elba Island, Italy

Christine Paulin-Mohring
http://www.lri.fr/~paulin/LASER

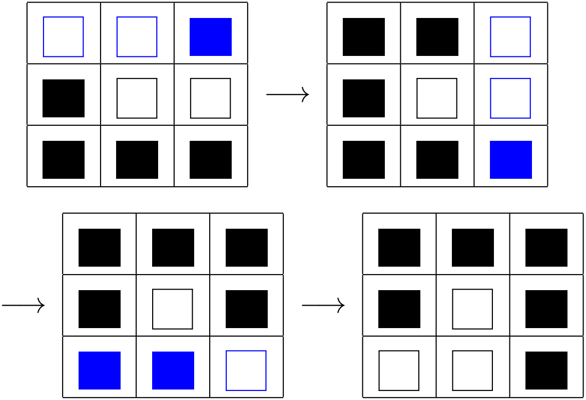
Université Paris Sud & INRIA Saclay - Île-de-France

September 2011

Lecture 3 : Functional programming with COQ

- Covers section 4 of course notes
- Example : `recursive.v`

Example



Reachability

`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.
```

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

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**.

Outline

- Introduction
- Basics of Coq system
- Using simple inductive definitions
- Functional programming with Coq
  - Recursive functions
  - Partiality and dependent types
  - Simple programs on lists
  - Well-founded induction and recursion
  - Imperative programming

(Co)Inductive definitions

- **match** only captures the fact that we have a fixpoint.
- any value in an inductive definitions starts with a constructor.
- consider the smallest or the greatest set of terms which satisfies this property.
- How to say that we have least or greatest fixpoints ?

Least fixpoints

Induction principle

```
Coq? Check nat_ind.
nat_ind : forall P : nat -> Prop,
    P 0 -> (forall n : nat, P n -> P (S n))
    -> forall n : nat, P n
```

Computational counterpart

```
Coq? Fixpoint leb (n m:nat) : bool :=
Coq?   match n with O => true
Coq?              | S p => match m with O => false
Coq?                                  | S q => leb p q
Coq?                       end
Coq?   end.
leb is recursively defined (decreasing on 1st argument)
```

Will not work is $\infty = \mathtt{S}\ \infty$ has type `nat`.

Fixpoint definitions

Fixpoint *name* $(x_1 : type_1) \ldots (x_p : type_p)\{\texttt{struct}\ x_i\}$
$: type_f \ := \ term.$

- Fixpoint is a term constructor
  (covers mutually recursive functions)
- Syntactic restriction on recursive calls on *term*.
- Sufficent for encoding many recursive functions, proving induction principles.
- Typing as usual for fixpoints.
- Reduction when x_i starts with a constructor.
- Equality always provable by case analysis.

Structural recursion

Only one argument decreases

```
Coq? Fixpoint test (b:bool) (n m:nat) :bool
Coq?  := match (n,m) with
Coq?     (O,_)    => true
Coq?   | (_,0)    => false
Coq?   | (S p,S q) => if b then test b p m else test b n q
Coq?  end.
Error: Cannot guess decreasing argument of fix.
```

Cannot be written directly like that.

Ackermann function

Structural recursion on higher-order functions:

```
Coq? Fixpoint ack (n m:nat) {struct n} : nat
Coq?  := match n with
Coq?     0    => S m
Coq?   | S p => let fix ackn (m:nat) {struct m} :=
Coq?                 match m with 0   => ack p 1
Coq?                            | S q => ack p (ackn q)
Coq?                 end
Coq?             in ackn m
Coq? end.
ack is recursively defined (decreasing on 1st argument)
```

Check equations are valid:

```
Coq? Lemma ack_Sn_0 : forall n, ack (S n) 0 = ack n 1.

Coq? Lemma ack_Sn_Sm : forall n m,
Coq?       ack (S n) (S m) = ack n (ack (S n) m).
```

are solved using `reflexivity`.

Empty inductive definitions

No constructor:

```
Coq? Inductive empty : Type := .
empty is defined
empty_rect is defined
empty_ind is defined
empty_rec is defined

Coq? Definition any A (e:empty) : A :=
Coq?     match e with end.
any is defined
```

No finite element:

```
Coq? Inductive E : Type := Ei : E -> E.
E is defined
E_rect is defined
E_ind is defined
E_rec is defined

Coq? Fixpoint Eany (A:Type) (x : E) : A :=
Coq?    match x with (Ei y) => Eany A y end.
Eany is recursively defined (decreasing on 2nd argument)
```