Program Semantics, **Verification**, and Construction

# Coq Proof Assistant

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MAP-i, Porto 2008

## Part II - Program Verification

- A Pratical Approach to the Coq Proof Assistant
  - Gallina Language
    - Syntax for Terms and Basic Commands
    - Inductive Definitinions
    - Function Definitions
    - Programming with Dependend Types
  - Tactics and Interactive Term/Proof Construction
    - Logical Reasoning in Coq
    - Axiom Declarations and Classical Reasoning
    - Induction and Inductive Predicates
- Small Demo
- Lab Session
- Case study: correctness of functional programs

## Bibliography

- Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, volume XXV of Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.
- Documentation of the coq proof assistant (version 8.1).
  - Tutorial (http://coq.inria.fr/V8.1/tutorial.html)
  - Reference Manual (http://coq.inria.fr/V8.1/refman/index.html)
  - Standard Library (http://coq.inria.fr/V8.1/stdlib/index.html)
  - FAQ (http://coq.inria.fr/V8.1/faq.html)
- Yves Bertot. Coq in a Hurry. http://arxiv.org/abs/cs/0603118
- Christine Paulin-Mohring and Jean-Christophe Filliâtre. Coq-lab (Types Summer School 2007). http://www.lri.fr/~paulin/TypesSummerSchool/lab.pdf
- Eduardo Giménez, Pierre Castéran. A Tutorial on Recursive Types in Coq. (http://www.labri.fr/Perso/~casteran/RecTutorial.pdf.gz)

## Pragmatics

- Installation
  - Binaries available for several operating systems (ms-windows, linux, macosx)
  - Local copy: http://gwlmf/mapi/Software/
  - On line documentation (local copy)
    - **Tutorial**: http://gwlmf.di.uminho.pt/mapi/doc/tutorial.html
    - **Reference Manual**: http://gwlmf.di.uminho.pt/mapi/doc/refman/
    - **Standard Library**: http://gwlmf.di.uminho.pt/mapi/doc/stdlib/
  - ...links from the course page.
- Several choices for user interaction:
  - **coqtop** - basic textual shell;
  - **coqide** - graphical interface;
  - **emacs proof-general mode** - powerful emacs mode offering the functionality available in coqide.

## Coq Proof Assistant

- The specification language of Coq is named Gallina. It allows to develop mathematical theories and to prove specifications of programs.
- It implements the Predicative Calculus of (Co)Inductive Definitions (pCiC).
- Every valid Gallina expression ”e” is associated to a type, or specification, ”t” (also a valid Gallina expression) - we denote the assertion “e has type t” by ”e:t”.
- Expressions in Gallina are built from:
  - pCiC terms - such as Sort names, abstractions, applications, ...
  - constants:
    - defined constants (name alias for other terms);
    - inductive types and associated objects (such as constructors, induction/recursion principles);
    - declarations - postulated inhabitants for certain types.
- A strong-normalising reduction relation is defined over the language expressions.
- These ingredients make it possible to look at the Coq system as:
  - **a Powerful Programming Language** (albeit some peculiarities, as we will see);
  - **a Proof Development Environment**.

# Coq as a Programming Language

# Coq as a Programming Language

- When seen as a programming language, the Coq system is capable to express most of the programs allowed in standard functional languages (e.g. haskell, SML, ...)
- Important distinctive features:
  - Consistency of the underlying theory relies on the strong-normalisation of terms. This enforces restrictions on:
    - recursion patterns allowed in the definitions;
    - shape on the type of constructors in inductive definitions
  - Ability to deal with dependent types, which allow a much finer notion of "types as specifications".
- Obviously, when we address the expressive power of Coq as a programming language, we are not interested in using it as the target language for an application - instead, we are interested in the expressive power that it accomplishes (for modelling purposes).
- This is particularly evident when verifying functional programs correctness - the objects we want to reason about are first class citizens in the proof environment.

# Definitions

- Coq allows to introduce new definitions, which link a name to a well-typed value.
- The impact of a definition is:
  - add a constant to the Gallina Language;
  - add a (δ-)redex to the evaluation rules.
- Coq commands:
  - **Definition** - define a new constant;
  - **Check** - asks the type of an expression;
  - **Print** - prints the definition of a constant;
  - **Eval** - evaluates an expression.

```
Coq < Definition double (x:nat) : nat := 2 * x.
double is defined
Coq < Check double.
double : nat -> nat
Coq < Print double.
double = fun x : nat => 2 * x
     : nat -> nat
Coq < Eval cbv delta [double] in (double 22).
     = (fun x : nat => 2 * x) 22
     : nat
Coq < Eval compute in (double 22).
     = 44 : nat
```

## Inductive Definitions

- The ability to define new types is an important ingredient for the expressiveness of a programming language.
- In Coq, some restrictions are imposed on the definable inductive types (c.f. the positivity condition presented earlier). However, the role of these ill-behaved types can arguably be referred as marginal in every-day programming.
- On the other hand, Coq allows for the definition of dependently typed inductive types, which are out of the scope in standard functional languages.
- As in functional languages, inductive types are specified by the signature of its constructors. Taking the following well known types as an example (haskell syntax):

```
 // Nil :: List a   Cons :: a -> List a -> List a
data List a = Nil | Cons a (List a)
// Empty :: Tree a;   Node :: Tree a -> a -> Tree a -> Tree a
 data Tree a = Empty | Node (Tree a) a (Tree a)
```

- In Coq, we must give the full signature of the constructors (return type included).

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
list, list_rect, list_ind, list_rec is defined
Inductive tree (A:Set) : Set :=
| empty : tree A
| node : tree A -> A -> tree A -> tree A.
tree, tree_rect, tree_ind, tree_rec is defined
```

## Induction/Recursion Principles

- The system automatically generates induction/recursion principles for the declared types.
- These allow the definition of primitive recursive functions on the respective type.
- Taking the list type as an example, primitive recursion is close enough to the "foldr" combinator available in the haskell prelude (slightly more powerful).
- The inferred principle is a dependent-type version of the primitive recursion combinator (return type might depend on the argument value). Its type is:

```
list_rec : forall (A : Set) (P : list A → Set),
        P nil →
        (forall (a :A) (l : list A), P l → P (a :: l)) →
        forall (l : list A), P l
```

- Sample functions:

```
Definition length' (A:Set) : list A -> nat :=
  list_rec A (fun _=>nat) 0 (fun x xs r=> 1+r).
length' is defined
Eval cbv delta beta iota in (length' (cons nat 1 (cons nat 1 (nil nat)))).
   = 2 : nat

Definition app' (A:Set) (l1 l2:list A) : list A :=
  list_rec A (fun _=>list A) l2 (fun x xs r=> cons A x r) l1.
app' is defined
Eval compute in (app' nat (cons nat 1 (nil nat)) (cons nat 2 (nil nat))).
  = cons nat 1 (cons nat 2 (nil nat)) : list nat
```

## Implicit Arguments

- The polymorphic lists example show that terms that, in a functional language are written simply as (cons 1 nil), are polluted with type applications (cons nat 1 (nil nat))
- These applications are annoying, since they can be inferred from the context.
- Coq has a mechanism that allow to omit these redundant arguments - **Implicit Arguments**.

```
Implicit Arguments nil [A].
Implicit Arguments cons [A].
Implicit Arguments length' [A].
Implicit Arguments app' [A].

Eval compute in (length' (cons 1 (cons 1 nil))).
     = 2 : nat

Eval compute in (app' (cons 1 nil) (cons 2 nil)).
     = cons 1 (cons 2 nil) : list nat
```

- The command **Set Implicit Arguments** instructs Coq to automatically detect what are the implicit arguments of a defined object.

## Fixpoint Definitions

- Fortunately, the Coq system allows for a direct encoding of recursive functions.

```
Fixpoint length (A:Set) (l:list A) : nat :=
  match l with
  | nil => 0
  | cons x xs => 1 + (length xs)
  end.
length is defined
Fixpoint app (A:Set) (l1 l2:list A) {struct l1} : list A :=
  match l1 with
  | nil => l2
  | cons x xs => cons x (app xs l2)
  end.
app is defined
```

- However, it forces recursive calls to act upon strict sub-terms of the argument (in the present of multiple arguments, the recursive one is singled out by the keyword "struct" - e.g. `{struct l1}`).
- A recent extension of the system allows to overcome this limitation...
  - ...as long as evidence for termination is provided (i.e. proved).

(...we will return to this later)

## Exploiting Dependent Types

- Dependent types allow for a much richer notion of type (e.g. we might express types for fixed-length lists; balanced-trees; binary-search trees; etc).

```
Inductive Vec (A:Set) : nat -> Set :=
| Vnil : Vec A 0
| Vcons : forall (x:A) (n:nat), Vec A n -> Vec A (S n).

Check (Vcons 1 (Vnil nat)).
Vcons 1 (Vnil nat) : Vec nat 1

Fixpoint appVec (A:Set)(n1 n2:nat)(v1:Vec A n1)(v2:Vec A n2){struct v1}:Vec A (n1+n2) :=
  match v1 in Vec _ m return Vec A (m+n2) with
   | Vnil => v2
   | Vcons x m' v' => Vcons x (appVec v' v2)
  end.

Eval compute in appVec (Vcons 1 (Vnil nat)) (Vcons 2 (Vcons 3 (Vnil nat))).
     = Vcons 1 (Vcons 2 (Vcons 3 (Vnil nat))) : Vec nat (1 + 2)
```

- The syntax in the "match" construct is extended to cope with different types in each branch and how the return type is conditioned.
- Dependent types lead to a proliferation of arguments (most of them inferable from others). Implicit arguments are an important tool to make the terms readable.
- Dependent types can be used to constrain the functional behaviour of programs - as such, their inhabitants might be seen as "programs with their correctness proof embedded on" (c.f. Σ-types as sub-set types - more about this later...).

- Programming with dependent types rapidly becomes a challenging task.
- Look at definition of the "head" function.

```
Fixpoint VecHead (A : Set) (n : nat) (v : Vec A (S n)) {struct v}: A :=
(match v in (Vec _ l) return (l <> 0 -> A) with
 | Vnil => fun h : 0 <> 0 => False_rec A (h (refl_equal 0))
 | Vcons x m' _ => fun _ : S m' <> 0 => x
 end)
  (* proof of Sn<>0 *)
  (fun H : S n = 0 =>
   let H0 :=
     eq_ind (S n)
       (fun e : nat => match e with
                       | 0 => False
                       | S _ => True
                       end) I 0 H in
   False_ind False H0)
```

- The difficulty arises from the fact that the first branch of the match constructor could not occur (Vnil is always of type (Vec A 0)). But the argument (proof) must be given explicitly.
- One often uses a mixed style of "definitional" and "interactive" term construction (more about this later).

# Coq as an Interactive Proof-Development Environment

## Interactive Proof Development

- The underlying type system used by coq allows for expressing proofs in the same way as we define functions acting on computational objects:

```
Definition simple_proof (A:Prop) (x:A) : A := x.
simple_proof is defined
Check simple_proof.
simple_proof : forall A : Prop, A -> A
```

- Here, we can identify the term (fun (A:Prop) (x:A) ⇒ x) as an encoding of the natural deduction proof tree for the logical formula (forall (A:Prop),A→A).
- However, when we are willing to find a proof for a given formula, the ability to present the complete proof-term is not very helpful (it can only act as a proof-checking device).
- To interactively construct the natural deduction proof-tree, is is normally adopted the so called goal-oriented proof development:
  - the system keeps track of the current goal and the set of open premises (the environment);
  - the user insert nodes in the proof-tree by means of certain commands (tactics). The system adjusts the state accordingly.
  - the process finishes when we are able to match a premise with the conclusion.

# Minimal Logic and Basic Tactics

- The internal logic of Coq possesses a single connective: $\Pi$. It generalises both implication and universal quantification.
- Basic tactics: **intro** - introduction rule for $\Pi$;
  **apply** - elimination rule for $\Pi$;
  **assumption**, **exact** - match conclusion with an hypothesis.
- Proof editing mode is activated by the **Theorem** command. Once completed, a proof is saved by the **Qed** command.

```
Theorem ex1 : forall A B:Prop, A -> (A->B) -> B.
1 subgoal
    ============================
   forall A B : Prop, A -> (A -> B) -> B
intros A B H H0.
1 subgoal
  A : Prop
  B : Prop
  H : A
  H0 : A -> B
  ============================
   B
apply H0.
1 subgoal
  A : Prop
  B : Prop
  H : A
  H0 : A -> B
  ============================
   A
exact H.
Proof completed.
Qed.
 ex1 is defined
```

# Curry-Howard analogy at work...

- Interactive goal-oriented proof-term construction is a legitimate mean to define objects in Gallina.

```
(* Definition Scomb (A B C:Set) (x:A->B->C) (y:A->B) (z:A) : C := x. *)
Definition Scomb (A B C:Set) : (A->B->C) -> (A->B) -> A -> C.
intros A B C x y z.
apply x.
  exact z.
  apply y; exact z.
Defined.
S is defined
```

- The command **Theorem** (and its variants **Lemma**, **Proposition**, **Corollary**, **Fact**, **Remark**) is analogous to **Definition** (but reserved for goal-oriented definitions).
- Note the use **Defined** command (instead of **Qed**). It makes the definition transparent (can be unfolded).
- The refine tactic allows to mix both styles - it fills a proof with a term (with holes).

```
Definition VecHead' (A:Set) (n:nat) (v:Vec A (S n)) : A.
refine (fun A n v => match v in Vec _ l return (l<>0)->A with
                     | Vnil => _
                     | Vcons x m' v => _
                     end _).
intro h; elim h; reflexivity.
intro h; exact x.
discriminate.
Defined.
```

# Logical Reasoning

- The other logical connectives are encoded as inductive types whose constructors correspond to their introduction rules:

```
(* False (absurd) is encoded as an empty type *)
Inductive False : Prop :=.
(* NEGATION - notation: ~A *)
Definition not (A:Prop) : Prop := A -> False.
(* AND - notation: A /\ B *)
Inductive and (A B:Prop) : Prop := conj : A -> B -> and A B.
(* OR - notation: A \/ B *)
Inductive or (A B:Prop) : Prop :=
| or_introl : A -> or A B
| or_intror : B -> or A B.
(* EXISTENCIAL QUANT. - notation “exists x, P x” *)
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P
```

- Elimination rules are obtained through the corresponding induction principle. Consider the following example:

```
Theorem ex2 : forall A B, A/\B -> A\/B.
intros; apply or_introl.
1 subgoal
  A : Prop
  B : Prop
  H : A /\ B
  ============================
   A
```

- To proceed the proof we have to apply the elimination rule on A/\B. Checking the type for and_ind, we get:

```
Check and_ind.
and_ind: forall A B P : Prop, (A->B->P) -> A/\B->P
```

- If we unify P with the conclusion, we can read it as a rule that decomposes the components of the conjunction.

```
apply and_ind with A B; try assumption.
1 subgoal
  A : Prop
  B : Prop
  H : A /\ B
  ============================
   A -> B -> A.
intros; assumption.
Proof completed.
```

(the “with” clause instantiate variables; the “try ...” tactical attempts to apply the given tactic to the generated sub-goals).

- The elim tactic performs exactly this:

```
Theorem ex2' : forall A B, A/\B -> A\/B.
intros A B H; left; elim H; intros; assumption.
Proof completed.
```

## Example: First-Order Reasoning

- The same methodology applies for proving first-order properties:

```
Theorem ex3 : forall (X:Set) (P:X->Prop), ~(exists x, P x) -> (forall x, ~(P x)).
unfold not; intros.
1 subgoal
  X : Set
  P : X -> Prop
  H : (exists x : X, P x) -> False
  x : X
  H0 : P x
  ============================
   False
apply H.
1 subgoal
  X : Set
  P : X -> Prop
  H : (exists x : X, P x) -> False
  x : X
  H0 : P x
  ============================
   exists x : X, P x
exists x (*apply ex_intro with x*); assumption.
Proof completed.
```

- **Exercise:** prove the reverse implication.

## Classical Reasoning

- The internal logic of Coq is constructive. That means that we could not directly prove certain classical propositions, such as:

(Peirce) ((P → Q) → P) → P

(Double negation elimination) ~~P → P

(de Morgan laws) ~ (forall n:U, ~ P n) → (exists n : U, P n)
~ (forall n:U, P n) → exists n : U, ~ P n.
~ (exists n : U, ~ P n) → forall n:U, P n

- To perform classical reasoning in Coq, we must rely on a convenient axiomatisation (e.g. adding the excluded middle principle as an axiom).
- In Coq, we might declare:

```
Axiom EM : forall P:Prop, P \/ ~P.
EM is assumed.
```

- Alternative names/commands for **Axiom** are **Conjecture** and **Parameter** (usually reserved for computational objects).

```
Theorem ex4 : forall (X:Set) (P:X->Prop), ~(forall x, ~(P x)) -> (exists x, P x).
intros.
1 subgoal
  A : Prop
  B : Prop
  X : Set
  P : X -> Prop
  H : ~ (forall x : X, ~ P x)
  ============================
   exists x : X, P x
elim (EM ((exists x, P x))).
2 subgoals
  A : Prop
  B : Prop
  X : Set
  P : X -> Prop
  H : ~ (forall x : X, ~ P x)
  ============================
   (exists x : X, P x) -> exists x : X, P x
subgoal 2 is:
 ~ (exists x : X, P x) -> exists x : X, P x
intro; assumption.
intro H0; elim H; red; intros; apply H0; exists x; assumption.
Proof completed.
```

- **Exercise**: prove the other formulas.

## Section mechanism

- The sectioning mechanism allows to organise a proof in structured sections.
- Inside these sections, the commands **Variable/Hypothesis** and **Let** declare local assumptions and definitions.
- When a section is closed, all local declarations (variables and local definitions) are discharged. This means that all global objects defined in the section are generalised with respect to all variables and local definitions it depends on in the section.

```
Section Test.
Variable U : Set.
Hypothesis Q R: U->Prop.
Theorem ex6: (forall x, Q x)/\(forall x, R x) -> (forall x, Q x /\ R x).
intro H; elim H; intros H1 H2; split; [apply H1 | apply H2]; assumption.
Qed.
Let conj (a b:Prop) := a/\b.
Hypothesis W Z:Prop.
Definition ex7 := conj W Z.
Check ex6.
ex6 : (forall x : U, Q x) /\ (forall x : U, R x) -> forall x : U, Q x /\ R x
Print ex7.
ex7 = conj W Z : Prop
End Test.
Check ex6.
ex6 : forall (U : Set) (Q R : U -> Prop),
  (forall x : U, Q x) /\ (forall x : U, R x) -> forall x : U, Q x /\ R x
Print ex7.
ex7 = let conj := fun a b : Prop => a /\ b in fun W Z : Prop => conj W Z
     : Prop -> Prop -> Prop
```

# Other uses of Axioms/Parameters

- Axioms and Parameters are often used to decouple the subject we are interested in reason about with some underlying theory (we comfortably take for granted).
- An example of such use is the modelling of an abstract data type - we are able to reason about it without depending on a concrete implementation.

```
Section Stack.

Variable U:Type.

Parameter stack : Type -> Type.
Parameter emptyS : stack U.
Parameter push : U -> stack U -> stack U.
Parameter pop : stack U -> stack U.
Parameter top : stack U -> U.
Parameter isEmpty : stack U -> Prop.

Axiom empty_isEmpty : isEmpty emptyS.
Axiom push_notEmpty : forall x s, ~isEmpty (push x s).
Axiom pop_push : forall x s, pop (push x s) = s.
Axiom top_push : forall x s, top (push x s) = x.

End Stack.

Check pop_push.
pop_push : forall (U : Type) (x : U) (s : stack U), pop (push x s) = s
```

# A note of caution!!!

- The capability to extend the underlying theory with arbitrary axioms is a powerful and dangerous mechanism.
- An axiom declaration turns the corresponding type automatically inhabited. Care must be taken in order to avoid inconsistency.
- As a simple demonstration of the risks, consider the following proof script:

```
Check False_ind.
False_ind : forall P : Prop, False -> P

Axiom ABSURD : False.
ABSURD is assumed.

Theorem ex8 : forall (P:Prop), P /\ ~P.
elim ABSURD.
Proof completed.
```

- The False_ind principle encodes the natural deduction ⊥-rule: from the absurd we are able to prove anything.
- Once declared an inhabitant for the False type, all the logical judgements are useless - the theory is inconsistent.

# Equality and Rewriting

- Equality is defined as an inductive predicate:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  refl_equal : x = x.

Check eq_ind.
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
   P x -> forall y : A, x = y -> P y
```

- Rewriting (replace equals by equals) is the natural proof strategy when dealing with equalities.

```
Lemma ex9  : forall a b c:X, a=b -> b=c -> a=c.
intros a b c H1 H2.
1 subgoal
  X : Set
  P : X -> Prop
  a,b,c : X
  H1 : a = b
  H2 : b = c
  ============================
   a = c
```

- In fact, rewriting is exactly what we achieve when perform elimination on a equality hypothesis...

```
...
elim H2 (*apply eq_ind with X b*).
1 subgoal
  X : Set
  P : X -> Prop
  a,b,c : X
  H1 : a = b
  H2 : b = c
  ============================
   a = b
elim H1.
  ...
  ============================
   a = a
reflexivity (*apply refl_equal*).
Proof completed.
```

- Equality tactics:
  - **rewrite** - rewrites an equality;
  - **rewrite <-** - reverse rewrite of an equality;
  - **reflexivity** - reflexivity property for equality;
  - **symmetry** - symmetry property for equality;
  - **transitivity** - transitivity property for equality.

## Evaluation and Convertibility tactics

- Another important proof strategy is evaluation.

```
Lemma app_nil : forall (A:Set) (l:list A), app nil l = l.
intros.
1 subgoal
  A : Set
  l : list A
  ============================
   app nil l = l
```

- To proceed the argument, it is enough to perform evaluation on the left hand side of the equality,

```
...
simpl.
1 subgoal
  A : Set
  l : list A
  ============================
   l = l
reflexivity.
Qed.
app_nil is defined
```

- Convertibility tactics:
  - **simpl, red, cbv, lazy** - performs evaluation;
  - **change** - replaces the goal by a convertible one.

## Induction (reasoning about inductive types)

- To reason about inductive types, we should apply the corresponding induction principle.
- For lists, it asserts that to prove a property (P l), is enough to prove:
  - P nil
  - forall x l, P l -> P (cons x l)

```
Theorem app_l_nil : forall (A:Set) (l:list A), app l nil = l.
intros A l; pattern l.
1 subgoals
  A : Type
  l : list A
  ============================
    (fun l0 : list A0 => l0 ++ nil = l0) l
apply list_ind.
2 subgoals
  A : Type
  l : list A
  ============================
   app nil nil = nil
subgoal 2 is:
 forall (a : A) (l0 : list A), app l0 nil = l0 -> app (a :: l0) nil = a :: l0
reflexivity.
intros a l0 IH; simpl; rewrite IH; reflexivity.
Proof completed.
```

(the **pattern** tactic performs eta-expansion on the goal - it makes explicit the predicate to be unified with the conclusion of the induction principle.)

## Tactics for induction

- Once again, the elim tactic might be used to apply the corresponding induction principle:

```
Theorem app_l_nil′ : forall (A:Set) (l:list A), app l nil = l.
intros A l; elim l.
reflexivity.
intros a l0 IH; simpl; rewrite IH; reflexivity.
Proof completed.
```

- Some available tatics:
  - **induction** - performs induction on an identifier;
  - **destruct** - case analysis;
  - **discriminate** - discriminates objects built from different constructors;
  - **injection** - constructors of inductive types are injections;
  - **inversion** - given an inductive type instance, find all the necessary condition that must hold on the arguments of its constructors.

## Inductive Predicates

- We have shown how to declare new inductively define Sets (datatypes). But we can also define relations (predicates).

```
Inductive Even : nat -> Prop :=
| Even_base : Even 0
| Even_step : forall n, Even n -> Even (S (S n)).
Even, Even_ind are defined
Inductive last (A:Set) (x:A) : list A -> Prop :=
| last_base : last x (cons x nil)
| last_step : forall l y, last x l -> last x (cons y l).
last, last_ind are defined
Check last.
last : forall A : Type, A -> list A -> Prop
```

- **Exercise**: use the inversion tactic to prove that forall x, ~(last x nil). Try to prove it avoiding that tactic (difficult).
- **Exercise**: define inductively the Odd predicate and prove that Even n => Odd (S n).
- We can also define mutually recursive inductive types:

```
Inductive EVEN : nat -> Prop :=
| EVEN_base : EVEN 0
| EVEN_step : forall n, ODD n -> EVEN (S n)
with ODD : nat -> Prop :=
| ODD_step : forall n, EVEN n -> ODD (S n).
EVEN, ODD are defined
EVEN_ind, ODD_ind are defined
```

## Libraries & Automatisation

- Proof development often take advantage from the large base of definitions and facts found in the Coq Standard Library, as well from some (limited, but very useful) forms of automatisation.
- Often used libraries:
  - **Arith** - unary integers;
  - **ZArith** - binary integers;
  - **List** - polymorphic lists;

```
Require Import List.

Check map.
map : forall A B : Type, (A -> B) -> list A -> list B
```

- Useful commands for finding theorems acting on a given identifier:
  - Search, SearchAbout, SearchPattern
- For some specific domains, Coq is able to support some degree of automatisation:
  - **auto** - automatically applies theorems from a database;
  - **tauto, intuition** - decision procedures for specific classes of goals (e.g. propositional logic);
  - **omega, ring** - specialized tactics for numerical properties.

## Other useful tactics and commands...

- Tactics:
  - **clear** - removes an hypothesis from the environment;
  - **generalize** - re-introduce an hypothesis into the goal;
  - **cut, assert** - proves the goal through an intermediate result;
  - **pattern** - performs eta-expansion on the goal.
- Commands:
  - **Admitted** - aborts the current proof (property is assumed);
  - **Set Implicit Arguments** - makes it possible to omit some arguments (when inferable by the system);
  - **Open Scope** - opens a syntax notation scope (constants, operators, etc.)
- See the Reference Manual...

# Small Demo...

# Problem Statement

- We are interested in proving a simple fact concerning the app function:

An element belonging to (app l1 l2) belongs necessarily to l1 or l2.

# A Logic-Oriented Approach

- We first shall define what is meant by “an element belongs to a list”

```
InL : A -> list A -> Prop
```

- Then we are able to state the property we want to prove...

```
Theorem InApp : forall (A:Type) (l1 l2:list A) (x:A), (InL x (app l1 l2)) <-> (InL x l1)\/(InL x l2)
```

# A Programming-Oriented Approach

- Another approach would be to program the “elem” function:

```
elem :: a -> [a] -> Bool
elem _ [] = false
elem x (y:ys) | x==y   = true
              | otherwise = elem x ys
```

- Now, the statement becomes:

```
Theorem ElemApp : forall (A:Type) (l1 l2:list A) (x:A), elem x (app l1 l2)=orb (elem x l1) (elem x l2)
```

  - “orb” is the boolean-or function.

# Filling the gap: correctness of “elem”

- We have seen two different approaches for the formalisation of a single property. What should be the preferred one?
  - The first approach makes fewer assumptions and is easier to follow;
  - The second rely on the behaviour of the functions “elem” and “orb”.
- If we formally relate the “InL” relation with the “elem” function, we fill the gap between both approachs.

```
Theorem InElem : forall (A:Type) (l:list A) (x:A), (InL x l) <-> (elem x l = true)
```

- In fact, we have just proved the correctness of elem:
  - The relation InL acts as its specification.

# Lab Session