Introduction to the Coq proof assistant
First part
A user point of view

Catherine Dubois[1]

[1]ENSIIE - CEDRIC, Évry, France

TOOLS 2011

Most examples used in the slides can be downloaded from the TOOLS web site:
`http://tools.ethz.ch/tools2011/download_area.htm`

Files tuto_part1.v, sort.v, simple_compiler.v (2nd part).

Coq in 2 slides

- Coq (means Calculus Of Constructions) is a proof assistant, developed since 1984, by INRIA (France)
  T.Coquand, G. Huet, C. Paulin, ..., B. Barras, H. Herbelin
- Coq allows the user to write specifications, (functional) programs and do proofs.
- Coq is based on a typed lambda-calculus with dependent types and inductive types (the richest in the Barendregt's cube).
- Coq can produce ML programs (extraction mechanism).
- Coq is also used as a back-end of other tools to verify proofs e.g. Focalize (`http://focalize.inria.fr/`) and CiME 3 (`http://a3pat.ensiie.fr`)
- Confidence ! Coq in Coq (B. Barras 1999)

Various applications (with *some* examples):

- Formalization of Mathematics
  - Constructive Mathematics (Nijmegen)
  - Four Color theorem (G. Gonthier, B. Werner)
  - Geometry (Strasbourg, INRIA Sophia-Antipolis)
- Program Verification
  - see the users'contributions: data structures and algorithms
- Security
  - Electronic banking protocols (Trusted Logic, Gemalto)
  - Mobile devices (Project Moebius)
- Programming languages semantics and tools
  - ML type inference (C. Dubois, 1999) - POPLmark challenge (mainly metatheory about functional calculi)
  - Formalisation of FJ, Java (subsets), JVM, Jakarta
  - CompCert C verified compiler (X. Leroy and al.)
  - Formal verification of the seL4 secure micro-kernel (G. Klein and al, NICTA)

Pragmatics

- Installation: Binaries available for several operating systems (ms-windows, linux, macosx), downloadable from the web page `http://coq.inria.fr/` (just coq in google: 1st entry)
- Several choices for user interaction:
  - coqtop - basic textual shell;
  - coqide - graphical interface;
  - emacs with proof-general mode - powerful emacs mode offering the functionality available in coqide.
- Through the Web: ProofWeb `http://prover.cs.ru.nl/`. Chose *coq* in the *provers* menu. OK with Firefox as a browser (better than safari e.g.)

Hands on if you want and have fun ... ☺

Coq often used for classical program verification:

- write (functional) programs,
- write their specifications,
- and then prove that the programs meet their specifications.

$\longrightarrow$ This view is taken as a roadmap for this tutorial

We can also mix programming and proving : more tricky to deal with dependent types
(see A. Chlipala's tutorial: An Introduction to Programming and Proving with Dependent Types in Coq. Journal of Formalized Reasoning, 2010)

Coq: A typed functional language

We can define functions (anonymous or named ones), apply them (partially or not), compose them.

Functions must be total, always terminate.

Functions are first class citizens.

No side effect, no exceptions

Some syntax:

$fun\ x : A \Rightarrow t$	with one argument
$fun\ (x : A)(y : B) \Rightarrow t$	with 2 arguments
$Definition\ f\,(x : A)(y : B) := t$	a named function
$f\ a$ or $f\,(a)$	application

- Assuming the type of natural numbers and the $+$ function:

```
Definition add3 (x: nat):= x + 3.
```

- Assuming the type of lists and the concatenation ++ function:

```
Definition at_end (l: list nat)(x: nat):= l++(x::nil).
```

- Higher order

```
Definition twice_nat (f: nat -> nat)(x: nat):= f (f x).
```

- (implicit) Polymorphism

```
Definition twice (A: Set)(f: A -> A)(x: A):= f (f x).
```

Everything is typed

```
Coq < Check add3.                    nat -> nat

Coq < Check 1 :: 2 :: 3 :: nil .     list nat

Coq < Check at_end.                  list nat -> nat -> list nat

Coq < Check twice.    forall A : Set, (A -> A) -> A -> A

Coq < Check nat.          Set

Coq < Check Set.          Type

Coq < Check forall A : Set, (A -> A) -> A -> A    Type

Coq < Check Type.                 Type
```

No paradox ! Type *denotes universes:* $Type\_i : Type\_i + 1$

Inductive Data Types

- Data types defined by constructors

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

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

- Pattern matching

```
Fixpoint plus (n m :nat) {struct n}: nat :=
  match n with
  | O => m
  | S p => S (plus p m)
end.
```

`plus` is a recursive function (structural recursion: call on a strict subterm of the structural argument, here `n`)
The `struct n` can be omitted (inferred).

```
Fixpoint foo (n :nat) struct n: nat :=
  match n with Coq   Error : recursive call to foo has
  | O => n           principal argument equal to
  | S p => foo (foo p)   "foo p" instead of "p"
end.
```

General recursion: not so easy: prove termination while defining

Some facilities with `Program` or `Function`.

```
Function f (x1:t1) ... (xn:tn) wf R xi : t =
body of the function
Proof .
Prove R is a well-founded relation
Prove each decreasing argument in a recursive call is
decreasing according to R
Qed.
```

Computation

We can compute within Coq (forget efficiency but it is not the subject!) with the command `Eval compute in ...`

```
Eval compute in (plus 2 3).
= 5 : nat
```

```
Eval compute in (twice nat pred 4).
= 2 : nat
```

```
Eval compute in (fun n => plus 0 n).
= fun n : nat => n : nat -> nat
```

You have in hands a typed pure functional language (core ML) with some more constraints: functions must terminate, pattern matching must be complete, more verbuous on types (in particular for polymorphic types - it can be relaxed in some places).
No mutable values.

Dependent types

Types depending on types

Parametric/generic types as in `forall A: Set, A -> A -> A`, `forall (A: Type), (list A)`

```
> Print list.
Inductive (A : Type) list : Type :=
nil : list A | cons : A -> list A -> list A
For nil : Argument A is implicit and maximally inserted
For cons : Argument A is implicit
...
```

$\Rightarrow$ Definition of a family of inductive types: `list nat`, `list bool`, `list (list nat)`, `list (nat -> nat)` ...

$\Rightarrow$ Implicit arguments: inferred in some places: a syntax close to ML (complete inference is impossible)

```
Definition add0 := fun l => cons 0 l.
> Check add0.
: list nat -> list nat

Same as :
Definition add0 := fun l : list nat => cons (A:=nat) 0 l.

Definition add := fun x => fun l => cons x l.
Error : Cannot infer the type of x

Definition add :=
   fun A : Set => fun x : A => fun l => cons x l.
```

⇒ Notations

- natural numbers: 0 is `O`, 1 is `S O`, 2 is `S (S O)`, etc.
- `a::tl` is `cons a tl`
- user-defined notations: to abbreviate and beautify specifications

```
Record rat : Set := {
top : nat ;
bot : nat
}.

Check (Build_rat 1 2).
Build_rat 1 2
     : rat
```

$\longrightarrow$ An inductive type with a unique constructor binary `Build_rat` and 2 accessors `top` and `bot`

```
Notation "p // q " := (Build_rat p q ) (at level 0).

Check (Build_rat 1 2).
(1) // (2)
     : rat

Eval compute in  (1 + 2) // 2.
     = (3) // (2)
     : rat

Definition mult_rat (r1 r2: rat) :=
let 'p1//q1:=r1 in
let 'p2//q2:= r2 in (p1*p2)//(q1*q2).

Notation "r1 '*r' r2":= (mult_rat r1 r2) (at level 0).

Eval compute in (4//5) *r (3//2).
     = (12) // (10)
     : rat
```

Dependent types

Types depending on values

A typical example: the type of vectors/arrays with a fixed length.

```
Inductive vect (A:Type) : nat -> Type :=
  Vnil : vect A 0
| Vcons : forall n, A -> vect A n -> vect A (S n).
```

Concatenation of vectors will have the type

```
forall A: Set, forall n m:  nat, vect A n -> vect
A m -> vect A (n+m).
```

Tricky to write functions on vectors.

Dependent types

Types depending on proofs

Let us redefine the type of rational numbers.

```
Record rat : Set := {
top : nat ;
bot : nat ;
wf : bot<>0      a proof that bot is not null
}
```

$\longrightarrow$ It is a dependent record (the 3rd field depends on the value of the snd field)

```
Check not_zero_4.
not_zero_4      : 4 <> 0

Check (Build_rat 1 4 with not_zero_4).
      : rat
```

$\longrightarrow$ mult_rat will now mix computations and proofs:

```
Check mult_non_0
mult_non_0
     : forall n m : nat, n <> 0 -> m <> 0 -> n * m <> 0
```

```
Notation "p / q 'with' H" := (Build_rat p q H) (at level 0)

Definition mult_rat (r1 r2: rat) :=
let 'p1/q1 with H1 :=r1 in
let 'p2/q2 with H2 := r2 in
(p1*p2)/(q1*q2) with (mult_non_0 q1 q2 H1 H2).
```

Coq: a language to specify

"Basics"

- `Prop`: type of properties

```
Coq < Check 0<1.                    : Prop
Coq < Check Prop.                   : Type
```

- `T -> Prop`: type of unary predicates

```
Coq < Check fun x => x > 0.         : nat -> Prop
```

- `T -> T' -> Prop`: type of binary predicates

```
Coq < Check fun x y => x-y >0.      : nat -> nat -> Prop
```

- Higher order logics (quantification on functions and predicates)

```
Definition Rel_refl (A: Set)(R : A -> A -> Prop) :=
forall x: A, R  x x .
forall A : Set, (A -> A -> Prop) -> Prop
```

Coq: a language to specify

Inductive Predicates

And also predicates defined inductively (rule-based definition)

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

- Each case of the definition is a theorem that lets you conclude `even(n)` appropriately.
- It is the smallest predicate that verifies both properties `even0` and `evenS`.
- The proposition `even(n)` holds only if it was derived by applying these theorems a finite number of times (smallest fixpoint).
- This is close to a Prolog definition.

Now Prove ...

- Prove that a function meets its specification e.g (or prove mathematical results)

```
forall l, l', l'=sort(l) -> sorted(l') /\ permut l l'
```

- How ?

  - automatically: generally not :-(

  - interactivly: drive the proof in a backward way with tactics

Tactics

Sequent presentation : $\dfrac{hypotheses}{goal}$

Backwards proofs directed by tactics (proof commands)

Basic tactics (natural deduction rules, derived rules, iota rules)

- `intro, intros`, `assumption`
- `split`, `decompose [and]`
- `left`, `right`, `decompose [or]`
- `exists`, `decompose [ex]`
- `assert`
- `simpl`
- etc.

More high level tactics

- `firstorder`
- `induction`, `cases`, `discriminate`, `inversion`
  (all trigged by reasoning principles associated to inductives types)
- `omega`, `ring`
- etc.

⚠ Intuitionistic logic (no excluded middle, no $\neg\neg$ elimination)

Basic Tactics

intro: introduction rule for -> and `forall`

$$\frac{H}{\text{forall}(x : t), A} \qquad \text{intro } x \ (x \text{ free in } H) \qquad \frac{\begin{array}{c} H \\ x : t \end{array}}{A}$$

or `intros` x_1 x_2 ... x_k to introduce several with the goal `forall`$(x_1 : t_1)(x_2 : t_2)...(x_n : t_n), A$

$$\frac{H}{A \text{ -> } B} \qquad \text{intro } H_1 \qquad \frac{\begin{array}{c} H \\ H_1 : t_1 \end{array}}{B}$$

or `intros` H_1 H_2 ... H_k to introduce several

Basic Tactics

assumption, exact, apply ...

$$\frac{H \quad x : A}{A}$$

assumption
exact x
apply x

```
proof (or subgoal)
completed
```

$$\frac{H \quad x : B \to A}{A}$$

apply x

$$\frac{H \quad x : B \to A}{B}$$

Basic Tactics

`left/right`, `decompose`: introduction, elimination $\vee$

$$\frac{H}{A \vee B}\ \text{left} \qquad \frac{H}{A}$$

$$\frac{H}{A \vee B}\ \text{right} \qquad \frac{H}{B}$$

$$\frac{H}{x : B \vee C}\ \text{decompose [or] } x \qquad \begin{cases} \dfrac{H}{y : B} \\ A \end{cases} \qquad \begin{cases} \dfrac{H}{y : C} \\ A \end{cases}$$

Basic Tactics

In action (with coqtop)

```
Lemma prf1 : forall p q : Prop, p /\ q -> q /\ p.
1 subgoal

  ============================
   forall p q : Prop, p /\ q -> q /\ p


prf1 < intros p q H.
1 subgoal

  p : Prop
  q : Prop
  H : p /\ q
  ============================
   q /\ p
```

```
prf1 < decompose [and] H.
1 subgoal

  p : Prop          q : Prop     H : p /\ q
  H0 : p            H1 : q
  ============================
   q /\ p

prf1 < split.
2 subgoals

  p : Prop          q : Prop     H : p /\ q
  H0 : p            H1 : q
  ============================
   q

subgoal 2 is:
 p                  (same hypothesis context than subgoal 1)
```

```
prf1 < assumption.     proof of subgoal 1
1 subgoal

 p : Prop          q : Prop     H : p /\ q
 H0 : p              H1 : q
 ============================
  p

prf1 < assumption.      proof of subgoal 2
Proof completed.

prf1 < Qed.
prf1 is defined
```

Proof script:

```
intros p q H.
decompose [and] H.
split.
 assumption.

 assumption.
```

or shorter

```
intros p q H.
decompose [and] H.
split; assumption.
```

; is a tactical
`tac1;tac2`: apply `tac2` on each subgoal generated by `tac1`
there are some more: `||` (applies tactic `t1` if it fails then applies `t2`) , `repeat`, etc.

On such a proof it can be fully automatic

```
Lemma prf1 : forall a b: Prop, a /\ b -> b /\ a.
prf1 <  firstorder.
proof completed.
```

A look at lambdas ...

prf1 is defined

- What's its type ? (`Check`)

```
forall p q : Prop, p /\ q -> q /\ p
```

- What's its value ? (`Print`)

```
fun (p q : Prop) (H : p /\ q) =>
and_ind (fun (H1 : p) (H2 : q) => conj H2 H1)
```

$\longrightarrow$ the proof is a lambda-term
The term can be rewritten as (with abusive notations):

```
fun (p q : Prop) (H : p /\ q) =>
match H with (H1, H2) => (H2, H1) end
```

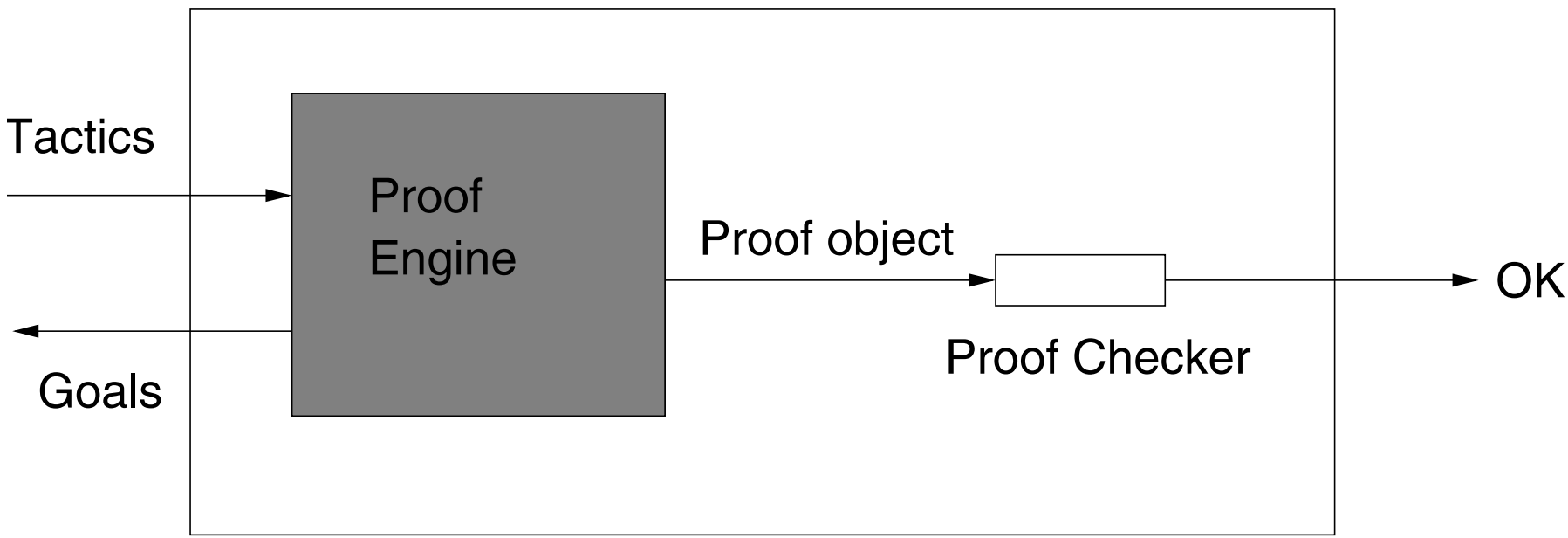
Curry-Howard isomorphism

Proofs as programs, types as specifications

- Read *t a type T* as *t is a proof of T*
- Read implications as functions and vice-versa: a function of type `A -> B` is a function that takes a proof of `A` as argument and computes a proof of `B`
- Read dependent types as quantifications
- etc.

- An interesting consequence of this analogy:

$$\text{proof checking} = \text{type checking}$$



Proofs by induction

double produces even numbers

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).

Fixpoint double (n:nat) :=
match n with
   0 => 0
| S p => S (S ( double p))
end.
```

double produces even numbers.

```
Lemma double_even: forall n, even (double n).
```

Proofs by induction

The reasoning principles

Let us go back to the definition of an inductive type.

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

```
Check nat_ind.          structural induction
: forall P : nat -> Prop,
P O -> (forall n, P n -> P (S n)) -> forall n, P n
Same for nat_rec (P: nat -> Set, recursor) and
nat_rect (P: nat -> Type).
```

The user can define and prove his own induction principles, if necessary.

Available for all inductive types, also for inductively defined predicates

$\Longrightarrow$ Reasoning principles: by case analysis on the last rule used; by induction on a derivation.

Proofs by induction

The example is back!

```
Coq < Lemma double_even: forall n, even(double n).
1 subgoal

  ============================
   forall n : nat, even (double n)

double_even < induction n.                    -- nat_ind is used
2 subgoals

  ============================
   even (double 0)

subgoal 2 is:
 even (double (S n))
```

```
double_even < simpl.
2 subgoals

  ============================
   even 0

double_even < apply even0.
1 subgoal

  n : nat   IHn : even (double n)
  ============================
   even (double (S n))

double_even < simpl.
1 subgoal

  n : nat    IHn : even (double n)
  ============================
   even (S (S (double n)))
```

```
double_even < apply evenS.
1 subgoal

  n : nat
  IHn : even (double n)
  ============================
   even (double n)

double_even < assumption.
Proof completed.
```

The next slides have not been presented during the conference.
The final case study (sorting by insertion) has been presented.

Another way

Mixing programs and specifications

Let us define `double` as a function from $\mathbb{N}$ to $2\mathbb{N}$.

- Type of even numbers defined as a subset (predicate subtype à la PVS) of `nat`: `{n: nat | even n}`
  (i.e. a nat and a proof this nat is even)

  Another to define vectors of length `n`:
  `{l: list nat | length l = n}`.

- `Program` : a way to manipulate more easily dependent types

```
Program Fixpoint double (n:nat):{p: nat | even(p)}:=
match n with
  0 => 0
| S p => S (S (double p))
end.
Coq < Solving obligations automatically...
2 obligations remaining
```

```
Obligation 1.
  ============================
   even 0
apply even0.
Defined.

Obligation 2.
double2 : nat -> {p : nat | even p}   p : nat
  ============================
   even (S (S (‘(double2 p))))

destruct (double2 p) as [y Hy].
  double2 : nat -> {p : nat | even p}    p : nat
  y : nat          Hy : even y
  ============================
   even (S (S (‘(exist (fun p0 : nat => even p0) y Hy))))
```

```
simpl.
double2 : nat -> {p : nat | even p}  p : nat
  y : nat          Hy : even y
  ============================
   even (S (S y))

apply evenS; assumption.
Defined.
```

More proof automation

auto - hints

- `auto` – tries a combination of tactics `intro`, `apply` and `assumption` using the theorems stored in a database as hints for this tactic. (Defined databases: core, arith, zarith, bool, etc.)
- `Hint Resolve/Rewrite` – add theorems to the database of hints to be used by auto using apply/rewrite.

```
Goal even(4).
auto.                                   no progress
repeat (apply evenS); apply even0.      Proof completed.

Hint Resolve even0.
Hint Resolve evenS.
```

Now the proof is done with auto.

$\longrightarrow$ Not too many `auto` and `hints` (proof scripts become uninformative)

More proof automation

Ltac

- Ltac: a programming language to define new tactics with matching operators for Coq terms but also proof contexts and backtracking

```
Ltac replace_le :=
match goal with
  H : (?X1 <= ?X2) |- _ => change (X2 >= X1) in H
| |- (?X1 <= ?X2) => change (X2 >= X1)
end.

Ltac replace_leR := repeat replace_le.
```

The tactic `replace_leR` replaces any `t1 <= t2` by `t2 >= t1` in the hypothesis context and in the conclusion.

The tactic in action:

```
a: nat                  q: nat
H0: 2 * q <= a
H1: a <= 2 * q + 1
====================
    S a <= 2 * S q + 1

.. > replace_leR.

a: nat                  q: nat
H0: a >= 2 * q
H1: 2 * q + 1 >= a
====================
      2 * S q + 1 >= S a
```

Program extraction

- Program extraction turns the informative contents of a Coq term into an ML program while removing the logical contents.
- Based on `Set` and `Prop` dinstinction.
- Different target languages: Ocaml, Haskell, Scheme.
- Confidence in the extraction mechanism ? a work in progress by P. Letouzey and S. Glondu, Paris 7

- Syntactic translation: `nat` and `double`
- Proofs are erased

```
Coq < Print rat.
Record rat : Set :=
  Build_rat { top : nat;  bot : nat;  wf : bot <> 0 }

Coq < Extraction rat.
type rat = { top : nat; bot : nat }

Coq < Extraction mult_rat.
(** val mult_rat : rat -> rat -> rat **)
let mult_rat r1 r2 =
  let { top = p1; bot = q1 } = r1 in
  let { top = p2; bot = q2 } = r2 in
  { top = (mult p1 p2); bot = (mult q1 q2) }
```

or `Recursive Extraction mult_rat.` (extraction of `nat`, `rat`, `mult`)

- A function can be extracted from a (constructive) proof

```
Coq < Lemma ex3 : forall n, m | n + n = m.
induction n.
exists 0.                 m=0
reflexivity.              prove 0+0=0
elim IHn.
intros t Ht.              if t is the witness for n, propose
exists (S (S t)).         S (S t) as a witness for S(n)
simpl.                    prove (S n) + (S n) = S (S t)
rewrite <- plus_n_Sm.
rewrite Ht; reflexivity.
Qed.

Coq < Extraction ex3.
(** val ex3 : nat -> nat **)
let rec ex3 = function
  | O -> O
  | S n0 -> S (S (ex3 n0))
```

Extraction from inductive predicates

Advertising !

- How to extract code from an inductive specification ?
- Out of the regular extraction mechanism that erases any `Prop`
- But some inductive specifications do have a computational content if we precise inputs/outputs.
- The user provides a mode = the list of the input positions. The extraction command extracts an ML function (without backtracking), may fail if the computation is considered as non deterministic.
- Not yet integrated in Coq, but expected to be a plugin in a next release
- Work done by D. Delahaye, P.N. Tollitte and myself, CEDRIC, ENSIIE, CNAM

```
Inductive add : nat -> nat -> nat -> Prop :=
| add0 : forall n, add n 0 n
| addS : forall n m p, add n m p -> add n (S m) (S p).
```

```
Mode {1,2} let rec add p0 p1 = match p0, p1 with
                               | n, O -> n
                               | n, S m -> let p = add n m in S
```

```
Mode {3,2} let rec add p0 p1 = match p0, p1 with
                               | n, O  -> n
                               | S p, S m  -> add p m
                               | _ -> assert false
```

```
Mode {1,2,3} let rec add p0 p1 p2 = match p0, p1, p2 with
                                    | n, O, m  -> if n=m then true
                                                  else false
                                    | n, S m, S p -> add n m p
                                    | _ -> false
```

```
Mode {1,3} Failure (non deterministic mode)
```

Typical usages for functional correctness:

- Define your ML function in Coq and prove it correct (and then extract back).
- Give the Coq function a richer type (a strong specification) and get the ML function via program extraction.
- Extract functions from inductive specifications, to prototype functions, to test specifications.

Insertion sort

A complete example: sorting a list of natural numbers in increasing order

$\Longrightarrow$ sorting by insertion - classical method (weak specification)

Insertion sort

The program

```
Check leb.                    leb   : nat -> nat -> bool

Fixpoint insert (x : nat) (l : list nat) :=
match l with
  nil => x::nil
| a::t =>  if leb x a then x::a::t else a::(insert x t)
end.

Fixpoint sort l :=
match l with
  nil => nil
| a::t => insert a (sort t)
end.

Eval compute in sort (1::4::3::0::(nil)).
     = 0 :: 1 :: 3 :: 4 :: nil      : list nat
```

Insertion sort

The specification

`sort` produces a sorted list containing the same elements than the initial list.

```
Inductive lelist : nat -> list nat -> Prop :=
 lenil: forall x, lelist x nil
|leS: forall x y l, x<=y -> lelist x l -> lelist x (y::l).

Inductive sorted : list nat -> Prop :=
 stnil : sorted nil
|stS : forall l x, lelist x l -> sorted l -> sorted (x::l).

Definition permut (l1 l2 : list nat) :=
   forall x:nat, count l1 x  = count l2 x.
```

with a function `count` that counts the number of occurrrences of an element in a list.

Insertion sort

The correctness proof

$\longrightarrow$ It relies on 2 theorems

1. `Lemma sort_sorted: forall l, sorted(sort l)`
   - By induction on `l`
   - `ins_corr: forall l x, sorted l -> sorted (insert x l)`
     - `le_lelist: forall l a x, le a x -> lelist x l -> lelist a l`
     - `lelist_ins: forall l a x , le a x -> lelist a l -> lelist a (insert x l)`

2. `Lemma sort_permut : forall l, permut l (sort l)`

- `permut_trans: forall l1 l2 l3, permut l1 l2 -> permut l2 l3 -> permut l1 l3`
- `permut_cons: forall a l1 l2, permut l1 l2 -> permut (a::l1) (a::l2)`
- `ins_permut: forall x l, permut (x::l) (insert x l)`

$\simeq$ 100 lines for spec & proof ($+$ 2 lemmas from the library) wrt 10 lines for the program