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.

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:

\[
\begin{align*}
\text{fun} & \quad x : A \Rightarrow t \quad \text{with one argument} \\
\text{fun} & \quad (x : A)(y : B) \Rightarrow t \quad \text{with 2 arguments} \\
\text{Definition} & \quad f(x : A)(y : B) := t \quad \text{a named function} \\

f \; a \text{ or } f(a) & \quad \text{application}
\end{align*}
\]
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

(implicit) Polymorphism
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
  | 0 => 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 x₁ : 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
...

⇒ Definition of a family of inductive types: list nat, list bool, list (list nat), list (nat -> nat) ...
⇒ 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 0, 1 is $S\ 0$, 2 is $S\ (S\ 0)$, etc.
- a::tl is cons a tl
- user-defined notations: to abbreviate and beautify specifications

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

Check (Build_rat 1 2).

```latex
Build_rat 1 2
  : rat
```

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
}

→ 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
mult_rat will now mix computations and proofs:

Check \texttt{mult\_non\_0}

\texttt{mult\_non\_0}

\hspace{1em} : \texttt{forall n m : nat, n <> 0 -> m <> 0 -> n * m <> 0}

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

\textbf{Definition \texttt{mult\_rat}} (r1 r2 : \texttt{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
  Coq < Check 0<1. : Prop
  Coq < Check Prop. : Type
  ```

- **T -> Prop**: type of unary predicates
  ```coq
  Coq < Check fun x => x > 0. : nat -> Prop
  ```

- **T -> T' -> Prop**: type of binary predicates
  ```coq
  Coq < Check fun x y => x-y >0. : nat -> nat -> Prop
  ```

- **Higher order logics** (quantification on functions and predicates)
  ```coq
  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)

\[
\text{Inductive even : nat \to Prop :=}
\begin{align*}
\text{even0 : even 0} \\
\text{| evenS : forall x:nat, even x \to even (S (S x)).}
\end{align*}
\]

- Each case of the definition is a theorem that lets you conclude \( \text{even}(n) \) appropriately.
- It is the smallest predicate that verifies both properties \( \text{even0} \) and \( \text{evenS} \).
- The proposition \( \text{even}(n) \) 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)
  \[ \forall l, l', l'=\text{sort}(l) \rightarrow \text{sorted}(l') \land \text{permut} \ l \ l' \]

- How?
  - automatically: generally not :-(
  - interactively: drive the proof in a backward way with tactics
Tactics

Sequent presentation: \( \text{hypotheses} \quad \frac{}{\text{goal}} \)

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 triggered by reasoning principles associated to inductives types)
- omega, ring
- etc.

⚠️ Intuitionistic logic (no excluded middle, no ¬¬ elimination)
Basic Tactics
intro: introduction rule for \( \rightarrow \) and \( \forall \)

\[
\begin{align*}
\frac{H}{\forall(x : t), \ A} & \quad \text{intro} \ x \\
\text{\( (x \ \text{free in} \ H \))} & \\
\frac{H}{\ A} & \quad \text{\( x : t \)}
\end{align*}
\]

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

\[
\begin{align*}
\frac{H}{A \rightarrow B} & \quad \text{intro} \ \ H_1 \\
\frac{H_1 : t_1}{B}
\end{align*}
\]

or intros \( H_1 \ H_2 \ldots H_k \) to introduce several
Basic Tactics
assumption, exact, apply ...

\[
\begin{align*}
H \\
\text{\hspace{1cm} } x : A \\
\downarrow \\
\text{\hspace{1cm} } A \\
\text{assumption} \\
\text{exact } x \\
\text{apply } x \\
\text{proof (or subgoal) completed}
\end{align*}
\]

\[
\begin{align*}
H \\
\text{\hspace{1cm} } x : B \to A \\
\downarrow \\
\text{\hspace{1cm} } A \\
\text{apply } x \\
\end{align*}
\]

\[
\begin{align*}
H \\
\text{\hspace{1cm} } x : B \to A \\
\downarrow \\
\text{\hspace{1cm} } B \\
\end{align*}
\]
Basic Tactics

left/right, decompose: introduction, elimination ∨

\[
\begin{align*}
\begin{array}{c}
H \\
\hline
A \lor B
\end{array} & \quad \text{left} \\
\begin{array}{c}
H \\
\hline
A \lor B
\end{array} & \quad \text{right} \\
\begin{array}{c}
H \\
\hline
x : B \lor C
\end{array} & \quad \text{decompose [or] } x \\
\begin{array}{c}
\hline
A
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
H \\
\hline
A
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
H \\
\hline
B
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
H \\
y : B \\
\hline
A
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
H \\
y : C \\
\hline
A
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\hline
A
\end{array}
\end{align*}
\]
Basic Tactics
In action (with coqtop)

Lemma prf1 : forall p q : Prop, p \land q \rightarrow q \land p.
1 subgoal

prf1 < intros p q H.
1 subgoal

p : Prop
q : Prop
H : p \land q

q \land p
prf1 < decompose [and] H.

1 subgoal

\[
\begin{align*}
p & : \text{Prop} & q & : \text{Prop} & H & : p \land q \\
H_0 & : p & H_1 & : q
\end{align*}
\]

\[
q \land p
\]

prf1 < split.

2 subgoals

\[
\begin{align*}
p & : \text{Prop} & q & : \text{Prop} & H & : p \land q \\
H_0 & : p & H_1 & : q
\end{align*}
\]

\[
q
\]

subgoal 2 is:

\[
p \quad (\text{same hypothesis context than subgoal 1})
\]
prf1 < assumption.  \textit{proof of subgoal 1}

1 subgoal

\begin{align*}
p & : \text{Prop} & q & : \text{Prop} & H & : p \land q \\
H0 & : p & H1 & : q \\
\end{align*}

\textbf{Proof completed.}

prf1 < Qed.

prf1 is defined
Proof script:

```coq
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)
  \[ \forall p \ q : \text{Prop}, \ p \land q \rightarrow q \land p \]

▶ What’s its value ? (Print)
  \[
  \text{fun} \ (p \ q : \text{Prop}) \ (H : p \land q) => \text{and}_{-}\text{ind} \ (\text{fun} \ (H1 : p) \ (H2 : q) => \text{conj} \ H2 \ H1) \\
  \rightarrow \text{the proof is a lambda-term} \\
  \text{The term can be rewritten as (with abusive notations):} \\
  \text{fun} \ (p \ q : \text{Prop}) \ (H : p \land q) => \text{match} \ H \ \text{with} \ (H1, H2) => (H2, H1) \ \text{end}
  \]
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 $A \rightarrow B$ 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:
proof checking = 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 :=
    0 : 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 0 -> (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

⇒ 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: $\{n: \text{nat} \mid \text{even } n\}$
  (i.e. a nat and a proof this nat is even)
Another to define vectors of length $n$:
$\{l: \text{list nat} \mid \text{length } l = n\}$.

- Program: a way to manipulate more easily dependent types
Program Fixpoint double (n:nat):$\{p: \text{nat} \mid \text{even}(p)\}$:=
match n with
  0 => 0
| S p => S (S (double p))
end.

Coq < Solving obligations automatically...
2 obligations remaining
Obligation 1.

\[
\begin{align*}
\text{even} \ 0 \\
\text{apply even}0. \\
\text{Defined.}
\end{align*}
\]

Obligation 2.

double2 : nat → \{p : nat | even p\}  \quad p : nat

\[
\begin{align*}
\text{even} \ (\text{S} \ (\text{S} \ (\text{double2} \ p)))
\end{align*}
\]
destruct (double2 p) as [y Hy].

double2 : nat → \{p : nat | even p\}  \quad p : nat

\[
\begin{align*}
y : \text{nat} & \quad Hy : \text{even} \ y \\
\text{even} \ (\text{S} \ (\text{S} \ (\text{exist} \ (\text{fun} \ p0 : \text{nat} \Rightarrow \text{even} \ p0) \ y \ Hy))))
\end{align*}
\]
simpl.

double2 : nat -> \{p : nat \mid \text{even } p\} \ p : nat
y : nat \quad \text{Hy : even } y

\begin{align*}
\text{even } (S (S \ y))
\end{align*}

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.

→ 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 \( t1 \leq t2 \) by \( t2 \geq t1 \) in the hypothesis context and in the conclusion.
The tactic in action:

\[ a : \text{nat} \quad q : \text{nat} \]
\[ \text{H0: } 2 \times q \leq a \]
\[ \text{H1: } a \leq 2 \times q + 1 \]
\[ \text{==================} \]
\[ S a \leq 2 \times S q + 1 \]

.. > replace_leR.

\[ a : \text{nat} \quad q : \text{nat} \]
\[ \text{H0: } a \geq 2 \times q \]
\[ \text{H1: } 2 \times q + 1 \geq a \]
\[ \text{==================} \]
\[ 2 \times S q + 1 \geq S a \]
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 distinction.
- 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.                \( \text{\#m=0} \)
  reflexivity.               \( \text{\#prove 0+0=0} \)
  elim IHn.
  intros t Ht.           \( \text{\#if t is the witness for n, propose} \)
  exists (S (S t)).      \( \text{\#S (S t) as a witness for S(n)} \)
  simpl.                  \( \text{\#prove (S n) + (S n) = S (S t)} \)
  rewrite <- plus_n_Sm.
  rewrite Ht; reflexivity.
  Qed.

Coq < Extraction ex3.

\[
(* * \text{val ex3 : nat \to nat **})
\]

\[
\text{let rec ex3 = function}
\]

\[
/ \ O \to \ O
\]

\[
/ \ S \ n0 \to \ S \ (S \ (ex3 \ n0))
\]
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 p

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
| 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

→ 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 occurrences of an element in a list.
Insertion sort
The correctness proof

→ It relies on 2 theorems

1. Lemma sort_sorted: \(\forall l, \text{sorted}(\text{sort} \ l)\)
  - By induction on \(l\)
  - ins_corr: \(\forall l \ x, \text{sorted} \ l \rightarrow \text{sorted} \ (\text{insert} \ x \ l)\)
    - le_lelist: \(\forall l \ a \ x, \text{le} \ a \ x \rightarrow \text{lelist} \ x \ l \rightarrow \text{lelist} \ a \ l\)
    - lelist_ins: \(\forall l \ a \ x, \text{le} \ a \ x \rightarrow \text{lelist} \ a \ l \rightarrow \text{lelist} \ a \ (\text{insert} \ x \ l)\)
2. Lemma `sort_permut` : \( \text{forall } l, \text{permut } l \text{ (sort } l) \)

- `permut_trans` : \( \text{forall } l1 \ l2 \ l3, \text{permut } l1 \ l2 \rightarrow \text{permut } l2 \ l3 \rightarrow \text{permut } l1 \ l3 \)

- `permut_cons` : \( \text{forall } a \ l1 \ l2, \text{permut } l1 \ l2 \rightarrow \text{permut } (a::l1) \ (a::l2) \)

- `ins_permut` : \( \text{forall } x \ l, \text{permut } (x::l) \ (\text{insert } x \ l) \)

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