Rigorous Software Development

Coq Proof Assistant

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MAP-i, Braga 2010

Part II - Deductive Reasoning

- A Practical Approach to the Coq Proof Assistant
  - Gallina Language
    - Syntax for Terms and Basic Commands
    - Inductive Definitions
    - Function Definitions
    - Programming with Dependent Types
  - Tactics and Interactive Term/Proof Construction
    - Logical Reasoning in Coq
    - Axiom Declarations and Classical Reasoning
    - Induction and Inductive Predicates
- Small Demo
- 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 8.3).
  - Tutorial (http://coq.inria.fr/tutorial.html)
  - Reference Manual (http://coq.inria.fr/refman/index.html)
  - Standard Library (http://coq.inria.fr/stdlib/index.html)
  - FAQ (http://coq.inria.fr/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)
  - ...downloadable from the web 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.

The 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 infer what are the implicit arguments in 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 O
| 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 device in making 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 (c.f. Curry-Howard Isomorphism)

```
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: Π. It generalises both implication and universal quantification.
- Basic tactics: **intro** - introduction rule for Π;
  apply - elimination rule for Π;
  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

- 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.
(* EXISTENTIAL 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 remaining example 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 $\bot$-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 tactics:
  - **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 conditions 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** - specialised 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 approaches.

```
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.

Program Verification in Coq

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

Part II - Program Verification

- A Pratical Approach to the Coq Proof Assistant
- Small Demo and Lab Session
- (Functional) Program Verification in Coq
  - Specifications and Implementations
    - correctness assertions
    - non-primitive-recursive functions in Coq
  - Functional Program Correctness
    - the direct approach
    - accurate types: specification-types and program-extraction
  - Case Study:
    - Verification of sorting programs

Coq as a Certified Program Development Environment

- From the very beginning, the Coq development team put a strong focus on the connection to program verification and certification.
- Concerning functional programs, we have already seen that:
  - it permits to encode most of the functions we might be interested in reason about - the programs;
  - its expressive power allows to express properties we want these programs to exhibit - their specifications;
  - the interactive proof-development environment helps to establish the bridge between these two worlds - correctness assurance.
- In the system distribution (standard library and user contributed formalisations) there are numerous examples of developments around the themes "certified algorithms" and "program verification".

Specifications and Implementations

Function Specifications

- What is a function specification?
  - In general, we can identify a function specification as a constraint on its input/output behaviour.
  - In practice, we will identify the specification of a function $f{:}A\to B$ as a binary relation $R\subseteq A\times B$ (or, equivalently, a binary predicate).
  - The relation associates each input to the set of possible outputs.
- Some remarks:
  - note that specifications do allow non-determinism (an element of the input can be related to multiple elements on the output) - this is an important ingredient, since it allows for a richer theory on them (composing, refinement, etc.);
  - it also means that doesn't exists a one-to-one relationship between specifications and functions (different functions can implement the same specification);
  - even when the specification is functional (every element of the domain type is mapped to exactly one element of the codomain), we might have different "functional programs" implementing the specification (mathematically, they encode the same function).

Partiality in Specifications

- Consider the empty relation $\emptyset \subseteq A \times B$ (nothing is related with anything). What is its meaning? Two interpretations are possible:
  - it is an "impossible" specification - it does not give any change to map domain values to anything;
  - it imposes no constrain on the implementation - thus, any function $f:A \to B$ trivially implements it.
- The second approach is often preferred (note that the first approach will make any non-total relation impossible to realise).
- So, we implicitly take the focus of the specification as the domain of the relation: a function $f:A \to B$ implements (realises) a specification $R \subseteq A \times B$ when, for every element $x \in dom(R)$, $(x,f(x)) \in R$. (obs.: dom(R) denotes the domain of R, i.e. $\{ a \mid (a,b) \in R \}$).
- The relation domain acts as a pre-condition to the specification.
- In practice, it is usually simpler to detach the pre-condition from the relation (by considering it a predicate Pre(-) on the domain type). The realisation assertion becomes:
  - for every element x of the domain type, $Pre(x) \Rightarrow (x,f(x)) \in R$.

Specification Examples

- Head of a list:

```
Definition headPre (A:Type) (l:list A) : Prop := l<>nil.

Inductive headRel (A:Type) (x:A) : list A -> Prop :=
  headIntro : forall l, headRel x (cons x l).
```

- Last element of a list:

```
Definition lastPre (A:Type) (l:list A) : Prop := l<>nil.

Inductive lastRel (A:Type) (x:A) : list A -> Prop :=
  lastIntro : forall l y, lastRel x l -> lastRel x (cons y l).
```

- Division:

```
Definition divPre (args:nat*nat) : Prop := (snd args)<>0.

Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
 let (n,d):=args in let (q,r):=res in q*d+r=n /\ r<d.
```

- Permutation of a list (example of a non-functional relation):

```
Definition PermRel (l1 l2:list Z) : Prop :=
  forall (z:Z), count z l1 = count z l
```

Implementations

- When we address the expressive power of Coq, we refer to some limitations in defining functions in Coq.
- But then, a question is in order: What is exactly the class of functions that can be encoded in Coq?
- The answer is: "functions provable total in higher-order logic".
- Intuitively, we can encode a function as long as we are able to prove it total in Coq.
- But the previous statement shouldn't be over emphasised! In practice, even if a function is expressible in Coq, it might be rather tricky to define it.
  - we can directly encode primitivive recursive functions (or, more generally, functions guarded by destructors);
  - Examples of functions that can not be directly encoded:
    - Partial functions;
    - non-structural recursion patterns (tricks and strategies...)
      - manipulate programs to fit the primitive-recursion scheme;
      - derive a specialised recursion principles;
      - Function command (available after Coq version V8.1).

Partial Functions

- Coq doesn't allow to define partial functions (function that give a run-time error on certain inputs)
- But Coq's type system allows to enrich the function domain with pre-conditions that assure that invalid inputs are excluded.
- Take the head (of a list) function as an example. In Haskell it can be defined as:

```
head :: [a] -> a
head (x:xs) = x
```

(the compiler exhibits a warning about "non-exhaustive pattern matching")

- In Coq, a direct attempt would fail:

```
Definition head (A:Type) (l:list A) : A :=
  match l with
  | cons x xs => x
  end.
Error: Non exhaustive pattern-matching: no clause found for pattern nil
```

- To overcome the above difficulty, we need to:
  - consider a precondition that excludes all the erroneous argument values;
  - pass to the function an additional argument: a proof that the precondition holds;
  - the match constructor return type is lifted to a function from a proof of the precondition to the result type.
  - any invalid branch in the match constructor leads to a logical contradiction (it violates the precondition).
- Formally, we lift the function from the type

  forall (x:A), B to forall (x:A), Pre x -> B
- Since we mix logical and computational arguments in the definition, it is a nice candidate to make use of the refine tactic...

```
Definition head (A:Type) (l:list A) (p:l<>nil) : A.
refine (fun A l p=>
  match l return (l<>nil->A) with
  | nil => fun H => _
  | cons x xs => fun H => x
  end p).
elim H; reflexivity.
Defined.
```

(the generated term that will fill the hole is “False_rect A (H (refl_equal nil))”)

- We can argue that the encoded function is different from the original.
- But, it is linked to the original in a very precise sense: if we discharge the logical content, we obtain the original function.
- Coq implements this mechanism of filtering the computational content from the objects - the so called extraction mechanism.

```
Check head.
head : forall (A : Type) (l : list A), l <> nil -> A

Extraction Language Haskell.
Extraction Inline False_rect.
Extraction head.

head :: (List a1) -> a1
head l =
  case l of
    Nil -> Prelude.error "absurd case"
    Cons x xs -> x
```

- Coq supports different target languages: Ocaml, Haskell, Scheme.

More on Extraction

- Coq's extraction mechanism are based on the distinction between sorts Prop and Set.
- ...but it enforces some restriction on the interplay between these sorts:
  - a computational object may depend on the existence of proofs of logical statements (c.f. partiality);
  - but the proof itself cannot influence the control structure of a computational object.
- As a illustrative example, consider the following function:

```
Definition or_to_bool (A B:Prop) (p:A\/B) : bool :=
  match p with
  | or_introl _ => true
  | or_intror _ => flase
  end.
Error:
Incorrect elimination of "p" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Set
because proofs can be eliminated only to build proofs.
```

- If we instead define a "strong" version of "or" connective, with sort Set (or Type):

```
Inductive sumbool (A B:Prop) : Type := (* notation {A}+{B} *)
| left : A -> sumbool A B
| right : B -> sumbool A B.
```

- Then, the equivalent of the previous function is:

```
Definition sumbool_to_bool (A B:Prop) (p:{A}+{B}) : bool :=
  match p with
  | left _ => true
  | right _ => flase
  end.
sumbool_to_bool is defined.

Extraction sumbool_to_bool.
sumbool_to_bool :: Sumbool -> Bool
sumbool_to_bool p =
  case p of
    Left -> True
    Right -> False
```

If - then - else -

- The sumbool type can either be seen as:
  - the or-connective defined on the Type universe;
  - or a boolean with logical justification embeded (note that the extraction of this type is isomorphic to Bool).
- The last observation suggests that it can be used to define an "if-then-else" construct in Coq.
  - Note that an expression like

    ```
    fun x y => if x<y then 0 then 1
    ```

    doesn't make sense: x<y is a Proposition - not a testable predicate (function with type X->X->bool);
  - Coq accepts the syntax

    ```
    if test then ... else ...
    ```

    (when test has either the type bool or {A}+{B}, with propositions A and B).
  - Its meaning is the pattern-matching

    ```
    match test with
      | left H => ...
      | right H => ...
    end.
    ```

- We can identify {P}+{~P} as the type of decidable predicates:
  - The standard library defines many useful predicates, e.g.
    - le_lt_dec : forall n m : nat, {n <= m} + {m < n}
    - Z_eq_dec : forall x y : Z, {x = y} + {x <> y}
    - Z_lt_ge_dec : forall x y : Z, {x < y} + {x >= y}
  - The command SearchPattern ({_}+{_}) searches the instances available in the library.
- Usage example: a function that checks if an element is in a list.

```
Fixpoint elem (x:Z) (l:list Z) {struct l}: bool :=
  match l with
    nil => false
  | cons a b => if Z_eq_dec x a then true else elem x b
  end.
```

- **Exercise:** prove the correctness/completeness of elem, i.e.

  forall (x:Z) (l:list Z), InL x l ↔ elem x l=true.

- **Exercise:** use the previous result to prove the decidability of InL, i.e.

  forall (x:Z) (l:list Z), {InL x l}+{~InL x l}.

Non obvious uses of the primitive recursion scheme

- Combining the use of recursors with higher-order types, it is possible to encode functions that are not primitive recursive.
- A well-known example is the Ackermann function.
- We illustrate this with the function that merges two sorted lists

```
merge :: [a] -> [a] -> a
merge [] l = l
merge (x:xs) [] = x:xs
merge (x:xs) (y:ys) | x <= y = x:(merge xs (y:ys))
                    | otherwise = y:(merge (x:xs) ys)
```

- In Coq, it can be defined with an auxiliary function merge':

```
Fixpoint merge (l1: list Z) {struct l1}: list Z -> list Z :=
  match l1 with
   | nil => fun (l2:list Z) => l2
   | cons x xs => fix merge' (l2:list Z) : list Z :=
                   match l2 with
                    | nil => (cons x xs)
                    | cons y ys => match Z_le_gt_dec x y with
                                    | left _ => cons x (merge xs (cons y ys))
                                    | right _ => cons y (merge' ys)
                                    end
                   end
  end.
```

Non-structural recursion

- When the recursion pattern of a function is not structural in the arguments, we are no longer able to directly use the derived recursors to define it.
- Consider the Euclidean Division algorithm,

```
div :: Int -> Int -> (Int,Int)
div n d | n < d = (0,n)
        | otherwise = let (q,r)=div (n-d) d
                      in (q+1,r)
```

- There are several strategies to encode these functions, e.g.:
  - consider an additional argument that “bounds” recursion (and then prove that, when conveniently initialised, it does not affect the result);

```
div :: Int -> Int -> (Int,Int)
div n d = divAux n n d
where divAux 0 _ _ = (0,0)
      divAux (x+1) n d | n < d = (0,n)
                       | otherwise = let (q,r)=divAux x (n-d) d
                                     in (q+1,r)
```

  (**Exercise:** define it in Coq and check its results for some arguments)

  - derive (prove) a specialised recursion principle.

Function command

- In recent versions of Coq (after v8.1), a new command **Function** allows to directly encode general recursive functions.
- The Function command accepts a measure function that specifies how the argument "decreases" between recursive function calls.
- It generates proof-obligations that must be checked to guaranty the termination.
- Returning to the div example:

```
Function div (p:nat*nat) {measure fst} : nat*nat :=
  match p with
  | (_,0) => (0,0)
  | (a,b) => if le_lt_dec b a
             then let (x,y):=div (a-b,b) in (1+x,y)
             else (0,a)
  end.
1 subgoal
  ============================
   forall (p : nat * nat) (a b : nat),
   p = (a, b) ->
   forall n : nat,
   b = S n ->
   forall anonymous : S n <= a,
   le_lt_dec (S n) a = left (a < S n) anonymous ->
   fst (a - S n, S n) < fst (a, S n)
```

- The proof obligation is a simple consequence of integer arithmetic facts (omega tactic is able to prove it).

```
intros; simpl.
omega.
Qed.
div_tcc is defined
div_terminate is defined
div_ind is defined
div_rec is defined
div_rect is defined
R_div_correct is defined
R_div_complete is defined
div is defined
div_equation is defined
```

- The Function command generates a lot of auxiliary results related to the defined function. Some of them are powerful tools to reason about it.
  - div_ind - a specialised induction principle tailored for the specific recursion pattern of the function (we will return to this later...)
  - div_equation - equation for rewriting directly the definition.
- **Exercise:** in the definition of the "div" function, we have included an additional base case. Why? Is it really necessary?

- The Function command is also useful to provide "natural encodings" of functions that otherwise would need to be expressed in a contrived manner.
- Returning to the "merge" function, it could be easily defined as:

```
Function merge2 (p:list Z*list Z)
{measure (fun p=>(length (fst p))+(length (snd p)))} : list Z :=
  match p with
  | (nil,l) => l
  | (l,nil) => l
  | (x::xs,y::ys) => if Z_lt_ge_dec x y
                     then x::(merge2 (xs,y::ys))
                     else y::(merge2 (x::xs,ys))
  end.
intros.
simpl; auto with arith.
intros.
simpl; auto with arith.
Qed.
```

- Once again, the proof obligations are consequence of simple arithmetic facts (and the definition of "length").
- As a nice side effect, we obtain an induction principle that will facilitate the task of proving theorems about "merge".

Functional Correctness

Direct approach

- Functional correctness establishes the link between a specification and an implementation.
- A direct approach to the correctness consists in:
  - Specification and implementation are both encoded as distinct Coq objects:
    - The specification is an appropriate relation (probably, with some predicate as precondition);
    - The implementation is a function defined in coq (probably with some “logical” precondition).
  - The correctness assertion consists in a theorem of the form:

    given a specification (relation fRel and a precondition fPre),
    a function f is said to be correct with respect to the specification if:

```
forall x, fPre x -> fRel x (f x)
```

div example

- Returning to our division function, its specification is:

```
Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
 let (n,d):=args in let (q,r):=res in q*d+r=n /\ r<d.

Definition divPre (args:nat*nat) : Prop := (snd args)<>0.
```

- The correctness is thus given by the following theorem:

```
Theorem div_correct : forall (p:nat*nat),  divPre p -> divRel p (div p).
unfold divPre, divRel.
intro p.
(* we make use of the specialised induction principle to conduct the proof... *)
functional induction (div p); simpl.
intro H; elim H; reflexivity.
(* a first trick: we expand (div (a-b,b)) in order to get rid of the let (q,r)=... *)
replace (div (a-b,b)) with (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
(* again a similar trick: we expand “x” and “y0” in order to use an hypothesis *)
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- e1.
omega.
(* and again... *)
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```

Function Completness

- Sometimes, we might be interested in a stronger link between the specifications and implementations.
- In particular, we might be interested in proving completeness - the implementation captures all the information contained in the specification:

```
forall x y, fPre x /\ fRel x y -> y=f x
```

- In this form, it can be deduced from correctness and functionality of fRel, i.e.

```
forall x y1 y2, fPre x /\ fRel x y1 /\ fRel x y2 -> y1=y2
```

- More interesting is the case of predicates implemented by binary functions. There exists a clear bi-directional implication. E.g.:

```
forall x l, InL x l <-> elem x l=true
```

Specification with Types

- Coq's type system allows to express specification constraints in the type of the function - we simply restrict the codomain type to those values satisfying the specification.
- This strategy explores the ability of Coq to express sub-types (Σ-types). These are defined as an inductive type:

```
(* Notation: { x:A | P x } *)
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
```

- Note that sig is a strong form of existential quantification (similar to the relation between or and sumbool).
- Using it, we can precisely specify a function by its type alone. Consider the type

  forall A (l:list A), l<>nil -> { x:A | last x l }

  (the last relation was shown earlier).
- Coq also defines

- Let us build an inhabitant of that type:

```
Theorem lastCorrect : forall (A:Type) (l:list A), l<>nil -> { x:A | last x l }.
induction l.
intro H; elim H; reflexivity.
intros.
destruct l.
exists a; auto.
assert ((a0::l)<>nil).
discriminate.
elim (IHl H0).
intros r Hr; exists r; auto.
Qed.
```

- And now, we can extract the computational content of the last theorem...

```
Extraction lastCorrect.

lastCorrect :: (List a1) -> a1
lastCorrect l =
  case l of
    Nil -> Prelude.error "absurd case"
    Cons a l0 ->
      (case l0 of
         Nil -> a
         Cons a0 l1 -> lastCorrect l0)
```

- This is precisely the "last" function as we would have written in Haskell.

Extraction approach summary

- When relying on the Coq's extraction mechanism, we:
  - exploit the expressive power of the type system to express specification constraints;
  - make **no distinction** (at least conceptually) between the activities of programming and proving. In fact, we build an inhabitant of a type that encapsulates both the function and its correctness proof.
- The extraction mechanism allows to recover the function, as it might be programmed in a functional language. Its correctness is implicit (relies on the soundness of the mechanism itself).
- Some deficiencies of the approach:
  - is targeted to "correct program derivation", rather than "program verification";
  - the programmer might lose control over the constructed program (e.g. a natural "proof-strategy" does not necessarily leads to an efficient program, use of sophisticated tactics, ...);
  - sometimes, it compromises reusing (e.g. proving independent properties for the same function).

Exercises

- Define a strong version of “elem”

```
elemStrong : forall (x:Z) (l:list Z), {InL x l}+{~InL x l}
```

  in such a way that its extraction is “analogous” (or uses) the elem function defined earlier.

- For the well known list functions app and rev provide:
  - a (relational) specification for them;
  - prove the correctness assertions.

Case Study: sorting functions

Sorting programs

- Sorting functions always give rise to interesting case studies:
  - their specifications is non trivial;
  - there are well-known implementations that achieve the expected behaviour through different strategies.
- Different implementations:
  - insertion sort
  - merge sort
  - quick sort
  - heap sort
- Specification - what is a sorting program?
  - computes a permutation of the input list
  - which is sorted.

Sorted Predicate

- A simple characterisation of sorted lists consists in requiring that two consecutive elements be compatible with the less-or-equal relation.
- In Coq, we are lead to the predicate:

```
Inductive Sorted : list Z -> Prop :=
  | sorted0 : Sorted nil
  | sorted1 : forall z:Z, Sorted (z :: nil)
  | sorted2 :
      forall (z1 z2:Z) (l:list Z),
        z1 <= z2 ->
        Sorted (z2 :: l) -> Sorted (z1 :: z2 :: l).
```

- **Aside:** there are other reasonable definitions for the Sorted predicate, e.g.

```
Inductive Sorted' : list Z -> Prop :=
  | sorted0' : Sorted nil
  | sorted2 :
      forall (z:Z) (l:list Z),
        (forall x, (InL x l) -> z<=x) -> Sorted (z :: l).
```

- The resulting induction principle is different. It can be viewed as a "different perspective" on the same concept.
- ...it is not uncommon to use multiple characterisations for a single concept (and prove them equivalent).

Permutation

- To capture permutations, instead of an inductive definition we will define the relation using an auxiliar function that count the number of occurrences of elements:

```
Fixpoint count (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0
  | (z' :: l') =>
      match Z_eq_dec z z' with
      | left _ => S (count z l')
      | right _ => count z l'
      end
  end.
```

- A list is a permutation of another when contains exactly the same number of occurrences (for each possible element):

```
Definition Perm (l1 l2:list Z) : Prop :=
  forall z, count z l1 = count z l2.
```

- **Exercise:** prove that Perm is an equivalence relation (i.e. is reflexive, symmetric and transitive).
- **Exercise:** prove the following lemma:

  forall x y l, Perm (x::y::l) (y::x::l)

insertion sort

- A simple strategy to sort a list consist in iterate an "insert" function that inserts an element in a sorted list.
- In haskell:

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys) | x<=y = x:y:ys
                | otherwise = y:(insert x ys)
```

- Both functions have a direct encoding in Coq.

```
Fixpoint insert (x:Z) (l:list Z) {struct l} : list Z :=
  match l with
    nil => cons x (@nil Z)
  | cons h t =>
        match Z_lt_ge_dec x h with
          left _ => cons x (cons h t)
        | right _ => cons h (insert x t)
        end
  end.
```

(similarly for isort...)

correctness proof

- The theorem we want to prove is:

```
Theorem isort_correct : forall (l l':list Z),
  l'=isort l -> Perm l l' /\ Sorted l'.
```

- We will certainly need auxiliary lemmas... Let us make a prospective proof attempt:

```
Theorem isort_correct : forall (l l':list Z),
  l'=isort l -> Perm l l' /\ Sorted l'.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl in H.
rewrite H.
1 subgoal
  a : Z
  l : list Z
  IHl : forall l' : list Z, l' = isort l -> Perm l l' /\ Sorted l'
  l' : list Z
  H : l' = insert a (isort l)
  ============================
   Perm (a :: l) (insert a (isort l)) /\ Sorted (insert a (isort l))
```

- It is now clear what are the needed lemmas:
  - insert_Sorted - relating Sorted and insert;
  - insert_Perm - relating Perm, cons and insert.

Auxiliary Lemmas

```
Lemma insert_Sorted : forall x l, Sorted l -> Sorted (insert x l).
Proof.
intros x l H; elim H; simpl; auto with zarith.
intro z; elim (Z_lt_ge_dec x z); intros.
auto with zarith.
auto with zarith.
intros z1 z2 l0 H0 H1.
elim (Z_lt_ge_dec x z2); elim (Z_lt_ge_dec x z1); auto with zarith.
Qed.

Lemma insert_Perm : forall x l, Perm (x::l) (insert x l).
Proof.
unfold Perm; induction l.
simpl; auto with zarith.
simpl insert; elim (Z_lt_ge_dec x a); auto with zarith.
intros; rewrite count_cons_cons.
pattern (x::l); simpl count; elim (Z_eq_dec z a); intros.
rewrite IHl; reflexivity.
apply IHl.
Qed.
```

Correctness Theorem

- Now we can conclude the proof of correctness...

```
Theorem isort_correct : forall (l l':list Z),
  l'=isort l -> Perm l l' /\ Sorted l'.
induction l; intros.
unfold Perm; rewrite H; split; auto.
simpl in H.
rewrite H.
elim (IHl (isort l)); intros; split.
apply Perm_trans with (a::isort l).
unfold Perm; intro z; simpl; elim (Z_eq_dec z a); intros; auto with zarith.
apply insert_Perm.
apply insert_Sorted; auto.
Qed.
```

Other sorting algorithms...

- We have proved the correctness of "insertion sort". What about other sorting algorithms like "merge sort" or "quick sort".
- From the point of view of Coq, they are certainly more challenging (and interesting)
  - their structure no longer follow a direct "inductive" argument;
  - we will need some auxiliary results...
- The first challenge is to encode the functions. E.g. for the merge sort, we need to encode in Coq the following programs:

```
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x<=y = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys

split [] = ([],[])
split (x:xs) = let (a,b)=split xs in (x:b,a)

merge_sort [] = []
merge_sort [x] = [x]
merge_sort l = let (a,b) = split l
               in merge (merge_sort a) (merge_sort b)
```

(here, the Function command is a big help!!!)

- Nice projects :-)