

More About Coq

Software Formal Verification

Maria João Frade

Departamento de Informática
Universidade do Minho

2008/2009

The Coq library

Proof development often take advantage from the large base of definitions and facts found in the Coq library.

- *The initial library*: it contains elementary logical notions and datatypes. It constitutes the basic state of the system directly available when running Coq.
- *The standard library*: general-purpose libraries containing various developments of Coq axiomatizations about sets, lists, sorting, arithmetic, etc. This library comes with the system and its modules are directly accessible through the `Require` command.

<http://coq.inria.fr/doc-eng.html>

- *Users' contributions*: user-provided libraries or developments are provided by Coq users' community. These libraries and developments are available for download.

<http://coq.inria.fr/contribs-eng.html>

Coq standard library

In the Coq system most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes.

Some often used packages:

<code>Logic</code>	Classical logic and dependent equality
<code>Arith</code>	Basic Peano arithmetic
<code>ZArith</code>	Basic relative integer arithmetic
<code>Bool</code>	Booleans (basic functions and results)
<code>Lists</code>	Polymorphic lists and Streams
<code>Sets</code>	Sets (classical, constructive, finite, infinite, power set, etc.)
<code>FSets</code>	Specification and implementations of finite sets and finite maps
<code>QArith</code>	Axiomatization of rational numbers
<code>Reals</code>	Formalization of real numbers
<code>Relations</code>	Relations (definitions and basic results)

...

Interpretation scopes

To simplify the input of expressions, the Coq system provides a notion of *interpretation scopes*, which define how notations are interpreted.

- An interpretation scope is a set of notations for terms with their interpretation.
- Interpretation scopes provides with a weak, purely syntactical form of notations overloading.
- Scopes may be opened and several scopes may be opened at a time.
- When a given notation has several interpretations, the most recently opened scope takes precedence (the collection of opened scopes may be viewed as a stack).
- It is possible to locally extend the interpretation scope stack using the syntax *(term)%key* (or simply *term%key* for atomic terms), where *key* is a special identifier called *delimiting key* and bound to a given scope.

Interpretation scopes

Check the following sequence of commands:

```
Require Import ZArith.
```

```
Require Import List.
```

```
Locate "_ * _".
```

```
Locate "_ + _".
```

```
Print Scope nat_scope.
```

```
Print Scope Z_scope.
```

```
Print Scope list_scope.
```

```
Check 3.
```

```
Eval compute in 4+5.
```

```
Check (3*5)%Z.
```

```
Eval compute in (3 + 5)%Z.
```

```
Open Scope Z_scope.
```

```
Check 3.
```

```
Check (S (S (S 0))).
```

```
Eval compute in 7*3.
```

```
Close Scope Z_scope.
```

```
Check 3.
```

Searching the environment

Some useful commands to find already existing proofs of facts in the environment.

- `Search ident` - displays the name and type of all theorems of the current context whose statement's conclusion has the form (*ident* t1 .. tn)
- `SearchAbout ident` - displays the name and type of all objects (theorems, axioms, etc) of the current context whose statement contains *ident*.
- `SearchPattern pattern` - displays the name and type of all theorems of the current context which matches the expression *pattern*.
- `SearchRewrite pattern` - displays the name and type of all theorems of the current context whose statement's conclusion is an equality of which one side matches the expression *pattern*.

Check the following commands:

```
Search le.
```

```
SearchAbout le.
```

```
SearchPattern (le (_ + _) (_ + _)).
```

```
SearchPattern (_ + _ <= _ + _).
```

```
SearchRewrite (_ + (_ - _)).
```

Implicit arguments

Some typing information in terms are redundant.

A subterm can be replaced by symbol `_` if it can be inferred from the other parts of the term during typing.

```
Definition comp : forall A B C:Set, (A->B) -> (B->C) -> A -> C
  := fun A B C f g x => g (f x).
```

```
Definition example (A:Set) (f:nat->A) := comp _ _ _ S f.
```

The implicit arguments mechanism makes possible to avoid `_` in Coq expressions. The arguments that could be inferred are automatically determined and declared as implicit arguments when a function is defined.

Set Implicit Arguments.

```
Definition comp1 : forall A B C:Set, (A->B) -> (B->C) -> A -> C
  := fun A B C f g x => g (f x).
```

```
Definition example1 (A:Set) (f:nat->A) := comp1 S f.
```

Implicit arguments

A special syntax (using @) allows to refer to the constant without implicit arguments.

```
Check (@comp1 nat nat nat S S).
```

It is also possible to specify an explicit value for an implicit argument.

```
Check (comp1 (C:=nat) S).
```

The generation of implicit arguments can be disabled with

```
Unset Implicit Arguments.
```

It is possible to enforce some implicit arguments.

```
Definition comp2 : forall A B C:Set, (A->B) -> (B->C) -> A -> C
:= fun A B C f g x => g (f x).
```

```
Implicit Arguments comp2 [A C].
```

```
Definition example2 (A:Set) (f:nat->A) := comp2 nat S f.
Print Implicit example2.
Print Implicit comp2.
```


Proof irrelevance

Let P be a proposition and t is a term of type P .

The following commands are **not** equivalent:

```
Theorem name : P.
```

```
Proof t.
```

```
Definition name : P := t.
```

- A definition made with `Definition` or `Let` is **transparent**: its value t and type P are both visible for later use.
- A definition made with `Theorem`, `Lemma`, etc., is **opaque**: only the type P and the existence of the value t are made visible for later use.
- Transparent definition can be unfolded and can be subject to δ -reduction, while opaque definitions cannot.

Basic tactics

- `intro`, `intros` – introduction rule for Π (several times)
- `apply` – elimination rule for Π
- `assumption` – match conclusion with an hypothesis
- `exact` – gives directly the exact proof term of the goal

Tactics for first-order reasoning

Proposition (P)	Introduction	Elimination (H of type P)
\perp		<code>elim H</code> , <code>contradiction</code>
$\neg A$	<code>intro</code>	<code>apply H</code>
$A \wedge B$	<code>split</code>	<code>elim H</code>
$A \Rightarrow B$	<code>intro</code>	<code>apply H</code>
$A \vee B$	<code>left</code> , <code>right</code>	<code>elim H</code>
$\forall x:A. Q$	<code>intro</code>	<code>apply H</code>
$\exists x:A. Q$	<code>exists</code> <i>witness</i>	<code>elim H</code>

Tactics for equational reasoning

- `rewrite` – rewrites a goal using an equality.
- `rewrite <-` – rewrites a goal using an equality in the reverse direction.
- `reflexivity` – reflexivity property for equality.
- `symmetry` – symmetry property for equality.
- `transitivity` – transitivity property for equality.
- `replace a with b` – replaces `a` by `b` while generating the subgoal `a=b`.
- ...

Convertibility tactics

- `simpl`, `red`, `cbv`, `lazy`, `compute` – performs evaluation.
- `unfold` – applies the δ rule for a transparent constant.
- `pattern` – performs a beta-expansion on the goal.
- `change` – replaces the goal by a convertible one.
- ...

Tactics for inductive reasoning

- `elim` – to apply the corresponding induction principle.
- `induction` – performs induction on an identifier.
- `case`, `destruct` – performs case analysis.
- `constructor` – applies to a goal such that the head of its conclusion is an inductive constant.
- `discriminate` – discriminates objects built from different constructors.
- `injection` – applies the fact that constructors of inductive types are injections.
- `inversion` – given an inductive type instance, find all the necessary condition that must hold on the arguments of its constructors
- ...

Other useful tactics and commands

- `clear` – removes an hypothesis from the environment.
- `generalize` – reintroduces an hypothesis into the goal.
- `cut`, `assert` – proves the goal through an intermediate result.
- `absurd` – applies False elimination.
- `contradict` – allows to manipulate negated hypothesis and goals.
- `refine` – allows to give an exact proof but still with some holes (“_”).
- ...

- `Admitted` – aborts the current proof and replaces the statement by an axiom that can be used in later proofs.
- `Abort` – aborts the current proof without saving anything.

Combining tactics

The basic tactics can be combined into more powerful tactics using tactics combinators, also called *tacticals*.

- `t1 ; t2` – applies tactic `t1` to the current goal and then `t2` to each generated subgoal.
- `t1 || t2` – applies tactic `t1`; if it fails then applies `t2`.
- `t ; [t1 | ... | tn]` – applies `t` and then `ti` to the *i*-th generated subgoals; there must be exactly *n* subgoals generated by `t`.
- `idtac` – does nothing
- `try t` – applies `t` if it does not fail; otherwise does nothing.
- `repeat t` – repeats `t` as long as it does not fail.
- `solve t` – applies `t` only if it solves the current goal.
- ...

The Coq system has a tactic language for programming new tactics: **Ltac**

Automatic tactics

- `trivial` – tries those tactics that can solve the goal in one step.
- `auto` – tries a combination of tactics `intro`, `apply` and `assumption` using the theorems stored in a database as hints for this tactic.
- `eauto` – like `auto` but more powerful but also more time-consuming.
- `autorewrite` – repeats rewriting with a collection of theorems, using these theorems always in the same direction.
- `tauto` – useful to prove facts that are tautologies in intuitionistic PL.
- `intuition` – useful to prove facts that are tautologies in intuitionistic FOL.
- `ring` – does proves of equality for expressions containing addition and multiplication.
- `omega` – proves systems of linear inequations (sums of $n * x$ terms).
- `field` – like `ring` but for a field structure (it also considers division).
- `fourier` – like `omega` but for real numbers.
- `subst` – replaces all the occurrences of a variable defined in the hypotheses.
- ...

Controlling automation

Several *hint databases* are defined in the Coq standard library. The actual content of a database is the collection of the hints declared to belong to this database in each of the various modules currently loaded.

- **Hint Resolve** – add theorems to the database of hints to be used by `auto` using `apply`.
- **Hint Rewrite** – add theorems to the database of hints to be used by `autorewrite`
- ...

Defined databases: `core`, `arith`, `zarith`, `bool`, `datatypes`, `sets`, `typeclass_instances`, `v62`.

One can optionally declare a hint database using the command `Create HintDb`. If a hint is added to an unknown database, it will be automatically created.

See the **Reference Manual** ...

Some datatypes of programming

```
Inductive unit : Set := tt : unit.
```

```
Inductive bool : Set := true : bool | false : bool.
```

```
Inductive nat : Set := 0 : nat | S : nat -> nat.
```

```
Inductive option (A : Type) : Type := Some : A -> option A  
                                     | None : option A.
```

```
Inductive identity (A : Type) (a : A) : A -> Type :=  
  refl_identity : identity A a a.
```

Some operations on `bool` are also provided: `andb` (with infix notation `&&`), `orb` (with infix notation `||`), `xorb`, `implb` and `negb`.

Some datatypes of programming

```
Inductive sum (A B : Type) : Type :=  
  | inl : A -> A + B  
  | inr : B -> A + B.
```

```
Inductive prod (A B : Type) : Type :=  
  pair : A -> B -> A * B.
```

```
Definition fst (A B : Type) (p : A * B) := let (x, _) := p in x.
```

```
Definition snd (A B : Type) (p : A * B) := let (_, y) := p in y.
```

The constructive sum $\{A\} + \{B\}$ of two propositions A and B.

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

If-then-else

- The `sumbool` type can be used to define an “if-then-else” construct in Coq.
- Coq accepts the syntax `if test then ... else ...` when `test` has either of 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}
```

If-then-else

A function that checks if an element is in a list.

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

Exercise:

Consider the inductive relation “y belongs to list l”

```
Inductive InL (A:Type) (y:A) : list A -> Prop :=  
| InHead : forall xs:list A, InL y (cons y xs)  
| InTail : forall (x:A) (xs:list A), InL y xs -> InL y (cons x xs).
```

and prove that

```
Theorem elem_corr : forall (a:Z) (l1 l2:list Z),  
  elem a (app l1 l2) = orb (elem a l1) (elem a l2).
```

The “subset” type

- Coq’s type system allows to combine a datatype and a predicate over this type, creating “the type of data that satisfies the predicate” . Intuitively, the type one obtains represents a **subset** of the initial type.

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=  
  exist : forall x : A, P x -> sig A P.
```

- Given $A:\text{Type}$ and $P:A\rightarrow\text{Prop}$, the syntactical convention for $(\text{Sig } A \ P)$ is the construct $\{x:A \mid P \ x\}$. (Predicate P is the *characteristic function* of this set).
- We may build elements of this set as $(\text{exist } x \ p)$ whenever we have a *witness* $x:A$ with its *justification* $p:P \ x$.
- From such a $(\text{exist } x \ p)$ we may in turn *extract* its witness $x:A$.
- In technical terms, one says that sig is a “*dependent sum*” or a Σ -*type*.

The “subset” type

A value of type $\{x:A \mid P\ x\}$ should contain a *computation* component that says how to obtain a value v and a *certificate*, a proof that v satisfies predicate P .

A variant `sig2` with two predicates is also provided.

```
Inductive sig2 (A : Type) (P Q : A -> Prop) : Type :=  
  exist2 : forall x : A, P x -> Q x -> sig2 A P Q
```

The notation for $(\text{sig2 } A\ P\ Q)$ is $\{x:A \mid P\ x \ \& \ Q\ x\}$.

Functional correctness

There are **two approaches** to defining functions and providing proofs that they satisfy given specification:

- To define these functions with a *weak specification* and then add *companion lemmas*.
For instance, we define a function $f : A \rightarrow B$ and we prove a statement of the form $\forall x : A, R x (f x)$, where R is a relation coding the intended input/output behaviour of the function.
- To give a *strong specification* to the function: the type of this function directly states that the input is a value x of type A and that the output is the combination of a value v of type B and a proof that v satisfies $R x v$.
This kind of specification usually relies on dependent types.

Partiality

The Coq system **does not allow** the definition of partial functions (i.e. functions that give a run-time error on certain inputs). However we can enrich the function domain with a precondition that assures that invalid inputs are excluded.

- A partial function from type A to type B can be described with a type of the form $\forall x:A, P x \rightarrow B$, where P is a predicate that describes the function's domain.
- Applying a function of this type requires two arguments: a **term** t of type A and a **proof** of the precondition $P t$.

An example

An attempt to define the head function as follows will 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).

An example

```
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.
```

Print Implicit head.

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

Arguments A, l are implicit

An example

The specification of head is:

```
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).
```

The correctness of function head is thus given by the following theorem:

```
Theorem head_correct : forall (A:Type) (l:list A) (p:headPre l),  
  headRel (head p) l.
```

Proof.

```
induction l.
```

```
intro H; elim H; reflexivity.
```

```
intros; destruct l; [simpl; constructor | simpl; constructor].
```

```
Qed.
```

Extraction

- Conventional programming languages do not provide dependent types and well-typed functions in Coq do not always correspond to well-typed functions in the target programming language.
- In CIC functions may contain subterms corresponding to proofs that have practically no interest with respect to the final value.
- The computations done in the proofs correspond to verifications that should be done once and for all at compile-time, while the computation on the actual data needs to be done for each value presented to functions at run-time.
- Coq implements this mechanism of filtering the computational content from the objects - the so called **extraction mechanism**.
- The distinction between the sorts Prop and Set is used to mark the logical aspects that should be discharged during extraction or the computational aspects that should be kept.

Extraction

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

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
```


Specification types

Using Σ -types we can express specification constraints in the type of a function - we simply restrict the codomain type to those values satisfying the specification.

- Consider the following definition of the inductive relation “x is the last element of list l”, and the theorem specifying the function that gives the last element of a list.

```
Inductive Last (A:Type) (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).
```

```
Theorem last_correct : forall (A:Type) (l:list A),  
l <> nil -> { x:A | Last x l }.
```

- By proving this theorem we build an inhabitant of this type, and then we can extract the computational content of this proof, and obtain a function that satisfies the specification.
- The Coq system thus provides a **certified software production tool**, since the extracted programs satisfy the specifications described in the formal developments.

Specification types

Let us build an inhabitant of that type

```
Theorem last_correct : forall (A:Type) (l:list A),  
                                l<>nil -> { x:A | Last x l }.
```

Proof.

```
induction l.
```

```
intro H; elim H; reflexivity.
```

```
intros. destruct l.
```

```
exists a; auto.
```

```
constructor.
```

```
elim IHl.
```

```
intros; exists x.
```

```
constructor. assumption.
```

```
discriminate.
```

```
Qed.
```

Program extraction

We can extract the computational content of the proof of the last theorem.

```
Extraction Language Haskell.
```

```
Extraction Inline False_rect.
```

```
Extraction Inline sig_rect.
```

```
Extraction Inline list_rect.
```

```
Extraction last_correct.
```

```
last_correct :: (List a1) -> a1
```

```
last_correct l =
```

```
  case l of
```

```
    Nil -> Prelude.error "absurd case"
```

```
    Cons a l0 -> (case l0 of
```

```
      Nil -> a
```

```
      Cons a0 l1 -> last_correct l0)
```

Case study: sorting a list

A simple characterisation of **sorted lists** consists in requiring that two consecutive elements be compatible with the \leq relation.

We can codify this with the following predicate:

```
Open Scope Z_scope.
```

```
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).
```

Case study: sorting a list

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

```
Fixpoint count (z:Z) (l:list Z) {struct l} : nat :=
  match l with
  | nil => 0%nat
  | (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.
```

Case study: sorting a list

Exercise:

Prove that Perm is an equivalence relation:

```
Lemma Perm_reflex : forall l:list Z, Perm l l.
```

```
Lemma Perm_sym : forall l1 l2, Perm l1 l2 -> Perm l2 l1.
```

```
Lemma Perm_trans : forall l1 l2 l3,  
    Perm l1 l2 -> Perm l2 l3 -> Perm l1 l3.
```

Exercise:

Prove the following lemmas:

```
Lemma Perm_cons : forall a l1 l2,  
    Perm l1 l2 -> Perm (a::l1) (a::l2).
```

```
Lemma Perm_cons_cons : forall x y l, Perm (x::y::l) (y::x::l).
```

Case study: sorting a list

A simple strategy to sort a list consist in iterate an “insert” function that inserts an element in a sorted list.

```
Fixpoint insert (x:Z) (l:list Z) {struct l} : list Z :=
  match l with
  | nil => cons x nil
  | 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.
```

```
Fixpoint isort (l:list Z) : list Z :=
  match l with
  | nil => nil
  | cons h t => insert h (isort t)
  end.
```

Case study: sorting a list

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. constructor. simpl in H.
```

```
rewrite H. (* ?????????????? *)
```

```
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))
```


Case study: sorting a list

It is now clear what are the needed lemmas:

```
Lemma insert_Perm : forall x l, Perm (x::l) (insert x l).
```

```
unfold Perm; induction l.
simpl. reflexivity.
simpl insert. elim (Z_lt_ge_dec x a). reflexivity.
intros. rewrite Perm_cons_cons.
pattern (x::l). simpl count. elim (Z_eq_dec z a).
intros. rewrite IHl; reflexivity.
intros. apply IHl.
Qed.
```

```
Lemma insert_Sorted : forall x l, Sorted l -> Sorted (insert x l).
```

```
intros x l H; elim H; simpl.
constructor.
intro z; elim (Z_lt_ge_dec x z); intros.
constructor.
auto with zarith.
...
Qed.
```

Case study: sorting a list

Now we can conclude the proof of correctness...

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

Proof.

```
induction l; intros.
```

```
unfold Perm; rewrite H; split; auto.
```

```
simpl. constructor. simpl in H.
```

```
rewrite H. (* ?????????????? *)
```

```
elim (IH1 (isort l)); intros; split.
```

```
apply Perm_trans with (a::isort l).
```

```
unfold Perm. intro z. simpl. elim (Z_eq_dec z a). intros.
```

```
elim H0; reflexivity.
```

```
intros. elim H0. reflexivity.
```

```
apply insert_Perm.
```

```
apply insert_Sorted.
```

```
assumption.
```

```
Qed.
```

Case study: sorting a list

Exercise:

Complete the following proof and extract its computational content to an Haskell function.

```
Definition inssort : forall (l:list Z),  
                        { l' | Perm l l' & Sorted l' }.
```

```
induction l.
```

```
...
```

```
Defined.
```

Exercise:

Use the same method to extract the insert function from its specification.

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 written in Haskell

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

- 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.

Non-structural recursion

```
Close Scope Z_scope.
```

```
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.
```

```
Proof.
```

```
intros.
```

```
simpl.
```

```
omega.
```

```
Qed.
```

The `Function` command generates a lot of auxiliary results related to the defined function. Some of them are powerful tools to reason about it.

Non-structural recursion

The `Function` command is also useful to provide “natural encodings” of functions that otherwise would need to be expressed in a contrived manner.

Exercise:

Complete the definition of the function `merge`, presenting a proof of its termination.

```
Function merge (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::(merge (xs,y::ys))
                      else y::(merge (x::xs,ys))
end.
```

Another example of correctness

A specification of the Euclidean division algorithm:

```
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.
```

A proof of correctness:

```
Theorem div_correct : forall (p:nat*nat), divPre p -> divRel p (div p).
```

```
Proof.
```

```
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.
```