Programming and Proving in Coq

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2021/2022

Roadmap

Programming and Proving in Coq

- some datatypes of programming;
- functional correctness; partiality; specification types;
- program extraction;
- non-structural recursion.

Some datatypes of programming

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

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

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

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


Exercises

Load the file `lesson3.v` in the Coq proof assistant to follow the examples of the coming slides. Analyse the examples and solve the exercises proposed.

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:

Inspect the proof of

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

and prove the following lemma:

```
Lemma ex : forall (a:Z) (l1 l2:list Z),
                        elem a (app l1 (cons a l2)) = true.
```


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:Type` and `P:A->Prop`, the syntactical convention for `(sig A P)` is the construct `{x:A | P x}`. (Predicate `P` is the *characteristic function* of this set).
- We may build elements of this set as `(exist x p)` whenever we have a witness `x:A` with its justification `p:(P x)`.
- From such a `(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 | 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 (sig2 A P Q) is {x:A | P x & Q x}.

Functional correctness

There are two approaches to define functions and provide proofs that they satisfy a given specification:

- To define these functions with a *weak specification* and then add *companion lemmas*.
  For instance, we define a function $f : A \to B$ 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* of 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 $\forall x : A, P\, x \to B$, 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 $P\, t$.

Example: the function head

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).

Example: the function head

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

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

Arguments A, l are implicit
```


Example: the function head

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 lemma:

```
Lemma head_correct : forall (A:Type) (l:list A) (p:headPre l),
                     headRel (head  p) l.
Proof.
  destruct 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 programing 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.
The extraction framework must be loaded explicitly.

```
Require Extraction.
```

```
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
```

Extraction

Extraction of all the mentioned objects and all their dependencies in the Coq toplevel.

```
Recursive Extraction head.
module Main where

import qualified Prelude

data List a =
   Nil
 | Cons a (List a)

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

Recursive extraction of all the mentioned objects and all their dependencies into a file.

```
Extraction "filename" head.
```


Extraction

The system also provides a mechanism to specify terms for inductive types and constructors of the target programming language.

For instance, we may want to use the Haskell native list type instead of the Coq one.

```
Extract Inductive list => "[]" [ "[]" "(:)" ].
```

```
Recursive Extraction head.
module Main where

import qualified Prelude

head :: ([] a1) -> a1
head l =
  case l of {
   [] -> Prelude.error "absurd case";
   (:) x _ -> x}
```


Specification types

Using Σ-types we can express specification constrains 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 specifing the function that gives the last element of a list.

```
Inductive Last (A:Type) (x:A) : list A -> Prop :=
| last_base : Last x (x :: nil)
| last_step : forall l y, Last x l -> Last x (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. 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 Inline False_rect.
Extraction Inline sig_rect.
Extraction Inline list_rect.
```

```
Recursive Extraction last_correct.
```

```
module Main where

import qualified Prelude

type Sig a = a
  -- singleton inductive, whose constructor was exist

last_correct :: ([] a1) -> a1
last_correct l =
  case l of {
   [] -> Prelude.error "absurd case";
   (:) y l0 -> case l0 of {
               [] -> y;
               (:) _ _ -> last_correct l0}}
```


Exercise

Exercise

Built an alternative definition of function head called “head_corr” based on the strong specification mechanism provided by Coq.
That is,

- give a strong specification of “head_corr”;
- prove it;
- and then, extract the computacional content of this proof.

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') => if Z.eq_dec z z'
                  then S (count z l')
                  else count z l'
  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

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.
```

Check the proofs.

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 => x :: nil
  | (h :: t) => if  Z.lt_ge_dec x h
                then x :: (h :: t)
                else h :: (insert x t)
  end.
```

```
Fixpoint isort (l:list Z) : list Z :=
  match l with
    nil => nil
  | (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:

```
Proof.
  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).

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

In order to prove them the following lemmas about count, may be useful.

```
Lemma count_insert_eq : forall x l,
                        count x (insert x l) = S (count x l).

Lemma count_cons_diff : forall z x l,
                        z <> x -> count z l = count z  (x :: l).

Lemma count_insert_diff : forall z x l,
                          z <> x -> count z l = count z (insert x l).
```

Check the proofs.

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 (IHl (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.
        -- auto with zarith.
      * 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.
  - exists nil. constructor. constructor.
  - elim IHl. intros. exists (insert a x).

    ...

Defined.
```


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)
```

- The 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.

Require Import Recdef. (* because of Function *)

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. lia.
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.
```


Introducing a new induction principle

After importing the module `List`, to use the standard notations for lists

```
Import ListNotations.
```

Consider the following splitting function.

```
Fixpoint split (A:Type) (l:list X) : (list A * list A) :=
  match l with
  | [] => ([],[])
  | [x] => ([x],[])
  | x1::x2::l' => let (l1,l2) := split l' in (x1::l1,x2::l2)
  end.
```

While this function is straightforward to define, it can be a bit challenging to work with.

Introducing a new induction principle

```
Lemma split_len_try: forall (A:Type) (l l1 l2: list A),
    split l = (l1,l2) -> length l1 <= length l /\ length l2 <= length l.
Proof.
  induction l; intros.
  - inversion H. simpl. lia.
  - destruct l as [| x l'].
    + inversion_clear H. split; simpl; auto.
    + inversion H. destruct (split l') as [l1' l2']. inversion H1.
Abort.
```

We're stuck!

```
 ...
 IHl : forall l1 l2 : list A, split (x :: l') = (l1, l2) ->
       length l1 <= length (x :: l') /\ length l2 <= length (x :: l')
 H : split (a :: x :: l') = (l1, l2)
 ============================
 length (a :: l1') <= length (a :: x :: l') /\
 length (x :: l2') <= length (a :: x :: l')
```

The IH talks about `split (x::l')` but we only know about `split (a::x::l')`.

Introducing a new induction principle

- The problem is that the standard induction principle for lists requires us to show that the property being proved follows for any non-empty list if it holds for the tail of that list.
- What we want here is a "two-step" induction principle, that instead requires us to show that the property being proved follows for a list of length at least two, if it holds for the tail of the tail of that list.

```
Definition list_ind2_principle:=
    forall (A : Type) (P : list A -> Prop),
      P nil ->
      (forall (a:A), P (a::nil)) ->
      (forall (a b : A) (l : list A), P l -> P (a :: b :: l)) ->
      forall l : list A, P l.
```

Introducing a new induction principle

If we assume the correctness of this non-standard induction principle, our proof is easy, using a form of the induction tactic that lets us specify the induction principle to use:

```
Lemma split_len': list_ind2_principle ->
    forall (A:Type) (l:list A) (l1 l2: list A),
    split l = (l1,l2) ->
    length l1 <= length l  /\  length l2 <= length l.
Proof.
  unfold list_ind2_principle; intro IP.
  induction l using IP; intros.
  - inversion H. lia.
  - inversion H. simpl; lia.
  - inversion H. destruct (split l) as [l1' l2']. inversion H1.
    simpl. destruct (IHl l1' l2') as [P1 P2]; auto; lia.
Qed.
```


Introducing a new induction principle

We still need to prove list_ind2_principle. There are several ways to do this. One direct way is to write an explicit proof term:

```
Definition list_ind2 :
  forall (A : Type) (P : list A -> Prop),
      P nil ->
      (forall (a:A), P (a::nil)) ->
      (forall (a b : A) (l : list A), P l -> P (a :: b :: l)) ->
      forall l : list A, P l :=
  fun (A : Type)
      (P : list A -> Prop)
      (H : P nil)
      (H0 : forall a : A, P (a::nil))
      (H1 : forall (a b : A) (l : list A), P l -> P (a :: b :: l))  =>
    fix IH (l : list A) :  P l :=
    match l with
    | nil => H
    | (x::nil) => H0 x
    | x::y::l' => H1 x y l' (IH l')
    end.
```


Introducing a new induction principle

With our "two-step" induction principle in hand, we can finally prove the lemma split_len.

```
Lemma split_len: forall (A:Type) (l:list A) (l1 l2: list A),
    split l = (l1,l2) ->
    length l1 <= length l  /\  length l2 <= length l.
Proof.
 apply (split_len' list_ind2).
Qed.
```


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. lia.
      * (* and again... *)
        change (snd (x,y0)<b); rewrite <- e1; assumption.
    + symmetry; apply surjective_pairing.
  - auto.
Qed.
```