Programming and Proving in Rocq

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2025/2026

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 Rocq.
- Rocq 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}
```

Defining notation

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.
```

We can define more convinent notation for decidable predicates. For example:

```
Notation "x == y" := (Z.eq_dec x y) (at level 70, no associativity).
Notation "x <? y" := (Z_lt_ge_dec x y) (at level 70, no associativity).

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


Exercises

Load the file `lesson3.v` in the Rocq Prover 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 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.
```


Σ-types (the "subset" types)

Rocq'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*.

Σ-types (the "subset" types)

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:

- Define these functions with a **weak specification** and then add **companion lemmas**.
  - 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.
- 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 $R\, x\, v$.

Partiality

The Rocq 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 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.
```


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 Rocq 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.
```

Extraction

- Conventional programming languages lack dependent types, and well-typed functions in Rocq do not always translate directly to well-typed functions in the target language.
- Functions may include proof terms that are irrelevant to the final computed value. These proofs represent checks that should occur at compile time, while actual data computations happen at run time.
- Rocq removes this non-computational content through its **extraction mechanism**. The distinction between `Prop` and `Set` marks logical components to discard during extraction and computational components to retain.

Extraction

- The extraction framework must be loaded explicitly.

```
From Stdlib Require Extraction.
```

- Rocq supports different target languages: Ocaml, Haskell, Scheme, JSON.

```
Extraction Language Haskell.
```

- All the mentioned objects and all their dependencies in the Rocq toplevel are extracted. We can inline them to make code more readable.

```
Extraction Inline False_rect.
Extraction Inline sig_rect.
Extraction Inline list_rect.
```

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

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


Extraction

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

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

- Recursive extraction of all the mentioned objects and all their dependencies.

```
Recursive Extraction last_correct.
```

- Recursive extraction into a file.

```
Extraction "exemplo1" last_correct.
```


Extraction

We can also convert the previously defined `head` function to Haskell.

```
Recursive Extraction head.
module Main where

import qualified Prelude

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

Exercise

Exercise

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

1. give a strong specification of "head_corr";
2. prove it;
3. and then, extract the computacional content of this proof.

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

The module `Recdef` of the standard library must be loaded.

```
From Stdlib 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.

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.
```