Program Verification in Coq

José Carlos Bacelar Almeida,
Jorge Sousa Pinto

Departamento de Informática
Universidade do Minho

Simão Melo de Sousa

Departamento de Informática
Universidade da Beira Interior

Program Verification

- (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
    - (Reasoning about non-terminating functions)

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 constrain 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 rich theory on them (composing, refinement, etc.);
  - it also means that doesn't exist 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 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 act as a pre-condition to the specification.
- In practice, it is usually simpler to detach the pre-condition from the relation (consider 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 the some limitations when 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 primitive-recursion scheme;
      - derive a specialised recursion principles;
      - Function command (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 constrains 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 in the last lecture).
- 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 constrains;
  - 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 :-)