Recursive functions in Coq

Yves Bertot

August 2011

Recursive datatypes

- Datatypes are described by several cases: the constructors
- Each constructor is presented as a function
  - Output type: the datatype
  - Inputs: they correspond to several fields
  - Some inputs are in the datatype: recursion
  - Usually one constructor has no inputs in the datatype
- Programming function with the datatype as input
  - Use `Match ... with ... end`
  - As many cases as there are constructors
  - One pattern variable for each non parameter constructor argument

An example of recursive datatype

- An example of datatype used to describe a programming language

```coq
Require Import String ZArith.
Inductive aexpr : Type :=
avar (x : string)
| anum (n : Z)
| aplus (e1 e2 : aexpr).
```

An example of recursive function

- Fixpoint `ev` with
  - Output type: `Z` (`string -> Z`)
  - Inputs: `aexpr`

```coq
Fixpoint ev (g : string -> Z) (e : aexpr):Z :=
match e with
| avar x => g x
| anum n => n
| aplus e1 e2 => ev g e1 + ev g e2
end.
```

- Note the recursive calls made on `e1` and `e2`

Proof techniques for recursive function

- Specific reasoning tool for each function
- Usable like an induction principle
  - Requires a special induction tactic
- Cases correspond to behavior cases of function
- Hypotheses are provided for each test performed
- Induction hypotheses are provided for recursive calls

Example of functional induction

- Fixpoint `fact` with
  - Output type: `nat`
  - Inputs: `x`

```coq
Fixpoint fact x :=
match x with
    O => 1
| S p => x * fact p
end.
```

- Functional Scheme `fact_ind` := Induction for `fact` Sort Prop.

```coq
Check fact_ind.
```

- `forall P : nat -> nat -> Prop,
  forall x : nat, x = 0 -> P 0 1
  (forall x p : nat,
    x = S p -> P p (fact p) -> P (S p) (x * fact p))
  forall x : nat, P x (fact x)
  ```
Second example of functional induction

Fixpoint div2 x :=
  match x with
    S (S p) => S (div2 p) | _ => 0 end.

Functional Scheme div2_ind := Induction for div2 Sort Prop.

Lemma div2te2e : forall x, div2 x * 2 <= x.
  intros x; functional induction div2 x.
  3 subgoals

Proof on div2

(* solve the first two easy cases *)
omega. omega.

More general recursive calls

▶ It is possible to have recursive calls on results of functions
▶ All cases must return a strict sub-term
▶ strict-subterms may be obtained by apply functions on strict subterms
  ▶ Only constraint is that functions must return sub-terms
  ▶ Not necessarily strict
  ▶ Checked by looking at all cases

Example of function that returns a subterm

Definition pred (n : nat) :=
  match n with
    O => n | S p => p end.

▶ in case O value is n, a (non-strict) sub-term of n
▶ in case S p value is n a sub-term of n

Recursive function using pred

Fixpoint sd (n : nat) :=
  match n with
    O => O
  | S p => S (sd (pred p))
  end.

▶ The same trick can be played with minus which returns a sub-term of its first argument, to define euclidian division
▶ To do as an exercise (using functional induction)

More general recursion

▶ Constraint of structural recursion too cumbersome
▶ Sometimes a characteristic decreases, but structural recursion is not available
▶ General solution provided by well-founded recursion
▶ Intermediate solution provided by the command Function
Example using Function

Require Import Recdef.

Function fact (x : Z) {measure Zabs x} :=
  if Zle_bool x 0 then 1 else x * fact (x - 1).
1 subgoal
============================
forall x : Z, Zle_bool x 0 = false ->
  (Zabs_nat (x - 1) < Zabs_nat x)%nat
SearchAbout Zle_bool. (* To find the right lemma *)
intros x tst; assert (cmp := Zle_cases x 0); rewrite tst in
  cmp : x > 0
SearchPattern (Zabs_nat _ < Zabs_nat _)%nat.
apply Zabs_nat_lt; omega.
Defined.

Comments on Function

- Integers have a more complex structure than natural numbers
  - \( x - 1 \) is not a structural subterm of \( x \)
  - For instance 3 is \( \text{Zpos (xI xH)} \) and 2 is \( \text{Zpos (xO xH)} \)
- Function but \( \text{Zabs_nat} \) provides a view on integers where a
subterm relation is present
- Makes more efficient computation possible
- Now, we prove explicitly that something decreases
- Finish the proof by Defined to make the function still evaluable
- Reason on functions by functional induction

Termination concerning two arguments

Definition mm (lls : list Z * list Z) :=
  let (l1, l2) := lls in length l1 + length l2.

Function merge (f:Z -> Z -> bool) (lls : list Z * list Z)
{measure mm lls} : list Z :=
  match lls with
  (nil, b) => b
  | (a::tl, nil) => a::tl
  | ((a::tl1) as l1, (b::tl2) as l2) =>
    if Zle_bool a b then
      a::merge f (tl1, l2)
    else
      b::merge f (l1, tl2)
  end.
intros; simpl; auto with arith.
Defined.

Proofs about uncurried functions

- Induction principles work for variables, no composite expressions
- All references to sub-components must be explicit references to
  the argument
- Two solutions to make references explicit
  - use projectors
  - add equalities

Example for merge

Lemma merge_perm :
  forall f x l1 l2,
  count x l1 count x l2 = count x (merge f (l1, l2)).
intros f x l1 l2.
(* using equalities *)
assert (H : forall u, u = (l1, l2) ->
  count x l1 + count x l2 =
  count x (merge f u)).
intros u; revert l1 l2; functional induction merge f l1 l2.

Example for merge

Lemma merge_perm :
  forall f x l1 l2,
  count x l1 count x l2 = count x (merge f (l1, l2)).
intros f x l1 l2.
(* using equalities *)
assert (H : forall u, u = (l1, l2) ->
  count x l1 + count x l2 =
  count x (merge f u)).
intros u; revert l1 l2; functional induction merge f u.

- revert l1 l2 is necessary: you discover it when using
  induction hypotheses
Mutual recursion

- Possible to define several inductive types at the same type
- Especially required when they rely mutually on eachother
- Associated induction is more complex

Inductive mtree (A : Type) : Type :=
mnode : A -> forest A -> mtree A
mcons : mtree A -> forest A -> forest A.

Mutually recursive function

Fixpoint occ A (p : A -> bool) (t : mtree A) : bool :=
  let (_ ,f) := t in
  match f with
  mnil => false
  | mcons t' f' => occ A p t' end.

Functional Scheme occ_ind := Induction for occ Sort Prop
with occf_ind := Induction for occf Sort Prop.

Proof about occ

Lemma occ_path : forall A (f : A -> bool) t,
  occ A f t = true -> exists l, exists v, exists ts,
  atpath A l t = Some (mnode A v ts) /
  f v = true.

Proof about occ

Fixpoint atpath A (l : list nat) t : option (mtree A) :=
  match l with
  nil => Some t
  | n::tl => let (_, f) := t in
    match atp A n f with
    None => None
    | Some t' => atpath A tl t'
  end.

Internal recursion

- Coq also makes it possible to describe anonymous recursive function
- Sometimes necessary to use them for difficult recursion patterns
- Especially when mixing polymorphic types inside type definitions

Inductive tree A : Type :=
  N (a : A) (l : list (tree A)).

Implicit Arguments N.
Example of function on tree

Require Import bool.

Fixpoint occurb A (f : A -> bool) t :=
    match t with
    N a l => f a ||
    (fix ol (l' : list (tree A)) :=
        match l' with
        nil => false
        | t1::ll => occurb A f t1 || ol ll
        end) l
    end.

Implicit Arguments occurb.

Compute occurb (beq_nat 3) (N 2 (N 4 nil::N 3 nil::nil)).
  = true : bool

Example of function on tree

Fixpoint collect A (t : tree A) :=
    match t with
    N a l => a ::
    (fix cl (l' : list (tree A)) :=
        match l' with
        nil => nil
        | t1::ll => collect A t1 ++ cl ll
        end) l
    end.

Implicit Arguments collect.

Compute collect (N 1 (N 2 (N 4 nil::N 3 nil::nil)::nil)).
  = 1::2::4::3::nil : list nat

Proofs about internal recursion

- Both induction and functional induction fail in this case
- You have to build your own induction principle
- Two recursive functions: a flavor of mutual recursion
- Proved by a function with a dependent type: a later lesson

forall A (P : tree A -> Prop) (Q : list (tree A) -> Prop)
(foreall a l, Q l -> P (N a l)) ->
(foreall t l, P t -> Q l -> Q (t::l)) ->
(Q nil) ->
forall t, P t