Recursive functions in Coq

Benjamin Grégoire

November 2011

Recursive datatypes

- Datatypes are described by several cases: the constructors
- Each constructor is presented as a function
  - Output type: the datatype
  - Inputs: they correpond to several fields
  - Some inputs are in the datatype: recursion
  - Usually one constructor has no inputs in the datatype (i.e. base cases)
- 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

```
Inductive btree : Type :=
| Empty : btree
| Node : btree -> btree -> btree.
```

An example of recursive function

```
Fixpoint get_subtree
   (l:list bool) (t:btree) {struct t} : btree :=
  match t, l with
  | Empty, _ => Empty
  | Node _ _, nil => t
  | Node tl tr, b :: l' =>
    if b then get_subtree l' tl else get_subtree l' tr
  end.
```

- Note the recursive calls made on `tl` and `tr`
- The recursive call should be done on strict sub-term
- This ensure the termination of recursive functions

Why termination is important

An Ocaml function:

```
# let rec loop x = loop x;;
val f : 'a -> 'b = <fun>
```

Why termination is important

Assume we can do such a definition in Coq:

```
Fixpoint loop (x:?) : ? := loop x.
```

Why termination is important

Now add type information for x:

```
Fixpoint loop (x:nat) : ? := loop x.
```

Why termination is important

Now add type information for the result (assume `B:Type`):

```
Fixpoint loop (x:nat) : B := loop x.
```

- The function `loop` has type `nat -> B`
- We have used `B` but we can use any type

Why termination is important

Use `False` instead of `B`:

```
Fixpoint loop (x:nat) : False := loop x.
```

- Now the function `loop` has type `nat -> False`
- Did you see the problem?

Why termination is important

Use `False` instead of `bool`:

```
Fixpoint loop (x:nat) : False := loop x.
```

- Now the function `loop` has type `nat -> False`
- Did you see the problem?
- What is the type of `loop 0` ?

Why termination is important

Use `False` instead of `bool`:

```
Fixpoint loop (x:nat) : False := loop x.
```

- Now the function `loop` has type `nat -> False`
- Did you see the problem?
- What is the type of `loop 0` ?

```
loop 0 :  False
```

- We can proof `False` without hypothesis, the logical system is incoherent (everything is provable)

The termination of recursive functions is one of the component which ensure the logical consistency of Coq
We should live with it . . .

An example of recursive function: `fact`

Recursive call should be made on strict sub-term:

```
Fixpoint fact n :=
  match n with
  | O => 1
  | S n' => n * fact n'
  end.

Definition fact' :=
  fix fact1 n :=
    match n with
    | O => 1
    | S n' => n * fact1 n'
    end.
```

An example of recursive function: div2

Recursive call can be done on not immediate sub-terms:

```
Fixpoint div2 n :=
  match n with
  | S (S n') => S (div2 n')
  | _ => 0
  end.
```

A sub-term of strict sub-term is a strict sub-term

More general recursive calls

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

Example of function that returns a sub-term

```
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 div2' (n : nat) :=
  match n with
    O => O
  | S p => S (div2' (pred p))
  end.
```

The same trick can be played with `minus` which returns a sub-term of its first argument, to define euclidian division

Mutual recursion

It is possible to define function by mutual recursion:

```
Fixpoint even n :=
  match n with
  | O => true
  | S n' => odd n'
  end
with odd n :=
  match n with
  | O => false
  | S n' => even n'
  end.
```

Lexicographic order

Sometimes termination functions is ensured by a lexicographic order on arguments (Ocaml):

```
let rec merge l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1::l1', x2::l2' ->
    if x1 <= x2 then
      x1 :: merge l1' l2
    else
      x2 :: merge l1 l2';;
```

There is two recursive call `merge l1' l2` and `merge l1 l2'`

Solution in Coq: internal recursion

Coq also makes it possible to describe *anonymous* recursive function
Sometimes necessary to use them for difficult recursion patterns

```
Fixpoint merge (l1 l2:list nat) : list nat :=
  match l1, l2 with
  | nil, _ => l2 |  _, nil => l1
  | x1::l1', x2::l2' =>
    if leb x1 x2 then x1::merge l1' l2
    else
      x2 :: (fix merge_aux (l2:list nat) :=
               match l2 with
               | nil => l1
               | x2::l2' =>
                 if leb x1 x2 then x1::merge l1' l2
                 else x2:: merge_aux l2'
               end) l2'
  end.
```

The style is a little bit boring (use the `Section` instead)

Another solution (Hugo Herbelin)

```
Fixpoint merge l1 l2 :=
  let fix merge_aux l2 :=
   match l1, l2 with
   | nil, _ => l2
   | _, nil => l1
   | x1::l1', x2::l2' =>
     if leb x1 x2 then x1::merge l1' l2
     else x2::merge_aux l2'
   end
  in merge_aux l2.

Compute merge (2::3::5::7::nil) (3::4::10::nil).
= 2 :: 3 :: 3 :: 4 :: 5 :: 7 :: 10 :: nil
     : list nat
```

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`: fact on Z

Integers have a more complex structure than natural numbers

```
Inductive positive : Set :=
  | xH : positive                  (* encoding of 1     *)
  | xO : positive -> positive      (* encoding of 2*p   *)
  | xI : positive -> positive.     (* encoding of 2*p+1 *)

Inductive Z : Set :=
  | Z0: Z | Zpos: positive -> Z | Zneg: positive -> Z.
```

- $x - 1$ is not a structural sub-term of x
- for instance 3 is `Zpos (xI xH)` and 2 is `Zpos (xO xH)`
- Makes more efficient computation possible

Example using `Function`: fact on Z

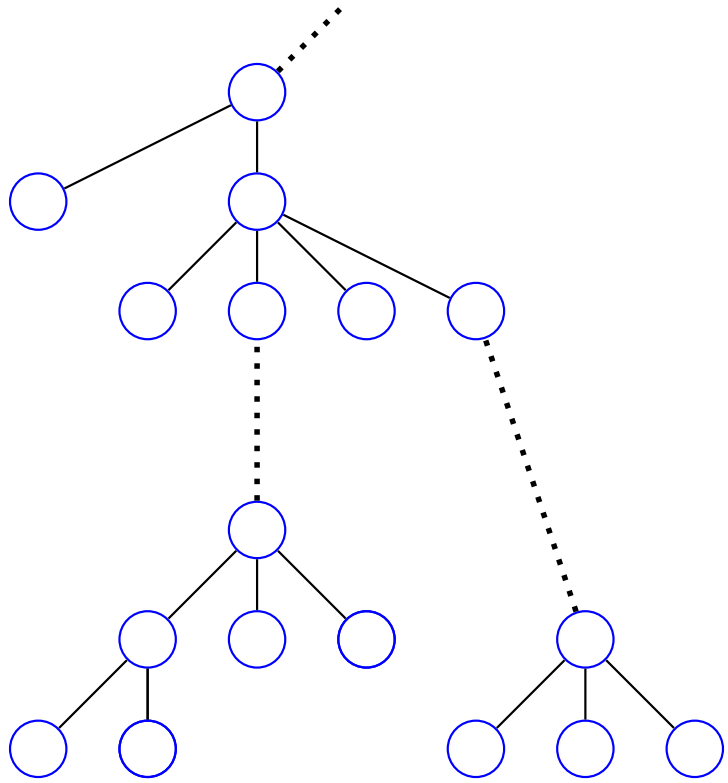
```
Require Import Recdef.

Function factZ (x : Z) {measure Zabs_nat 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
```

Now, we prove explicitely that something decreases

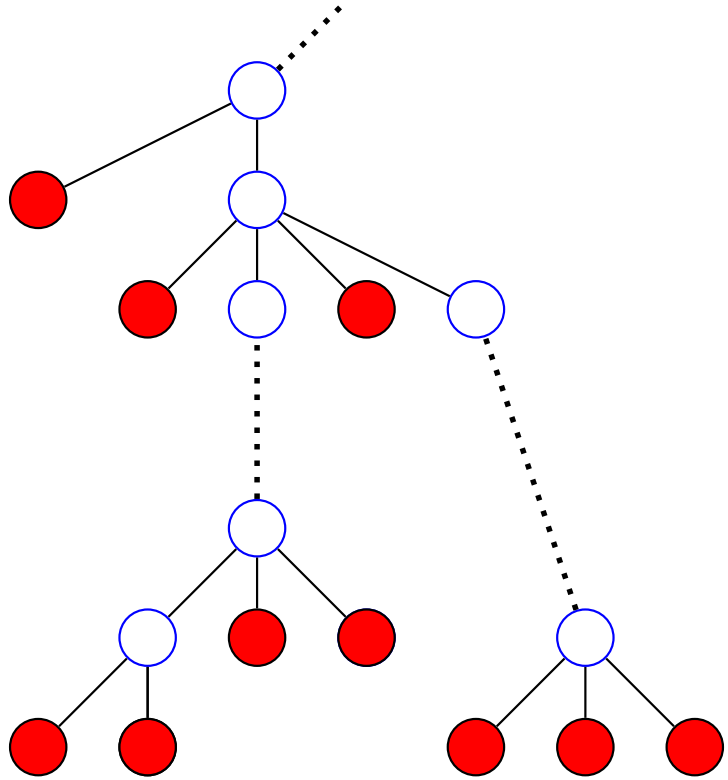
Merge again

```
Definition slen (p:list nat * list nat) :=
  length (fst p) + length (snd p).

Function Merge (p:list nat * list nat)
   { measure slen p } : list nat :=
  match p with
  | (nil, l2) => l2
  | (l1, nil) => l1
  | ((x1::l1') as l1, (x2::l2') as l2) =>
     if leb x1 x2 then x1::Merge (l1',l2)
     else x2::Merge (l1,l2')
   end.
(* Two goals *)
...
Defined.

Compute Merge (2::3::5::7::nil, 3::4::10::nil).
```

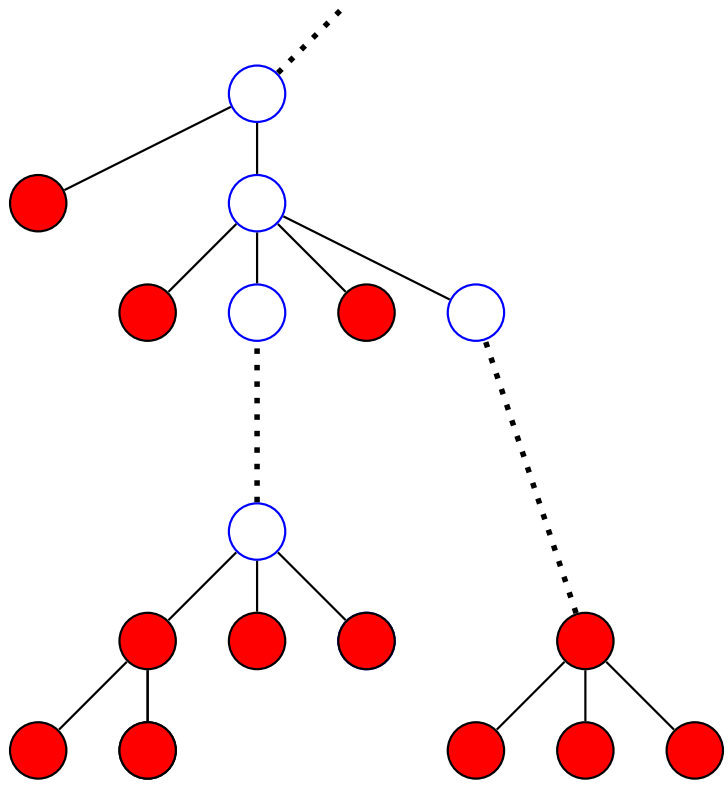
Well-founded Relations



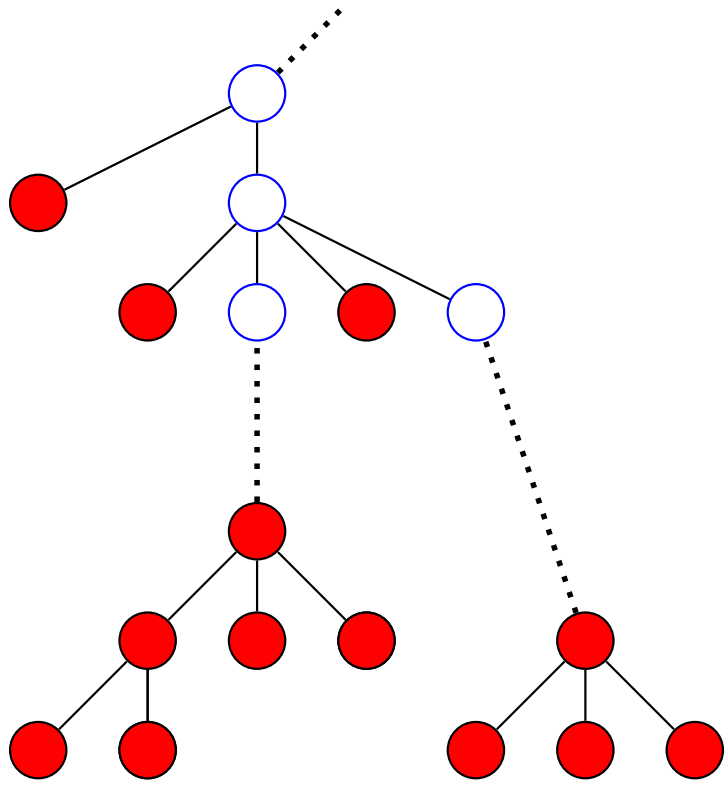
Dotted lines represent any number of elementary relationships



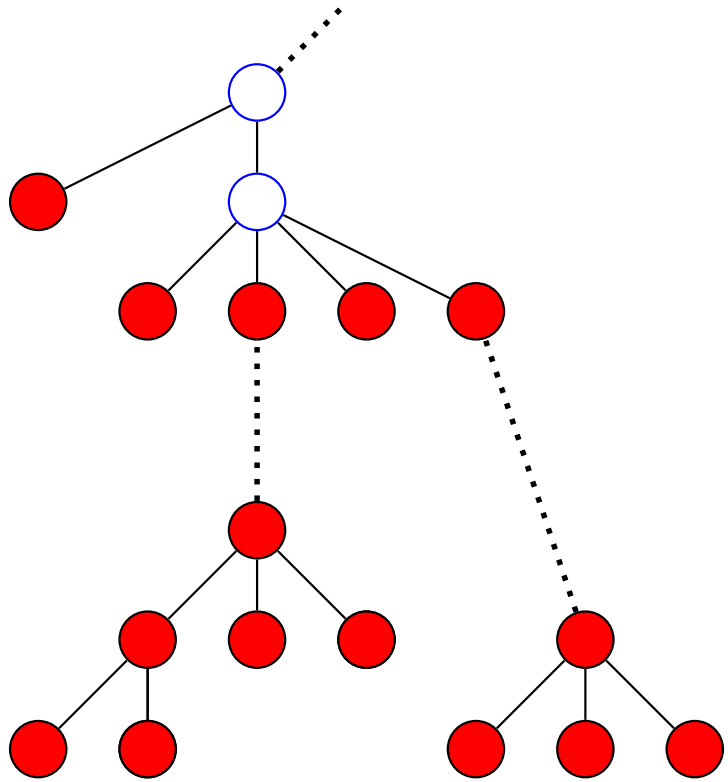
Minimal elements are *accessible*



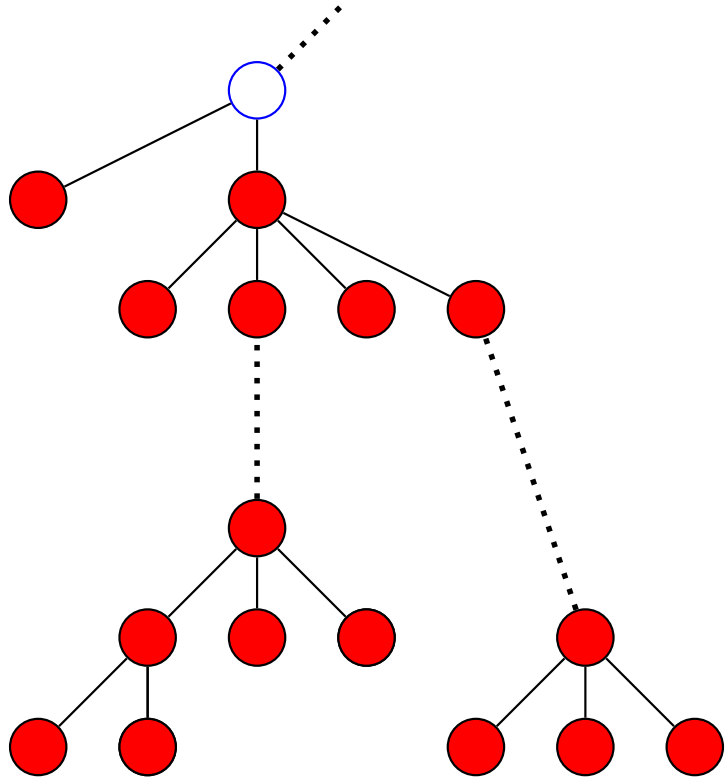
Elements whose all predecessors are accessible become accessible

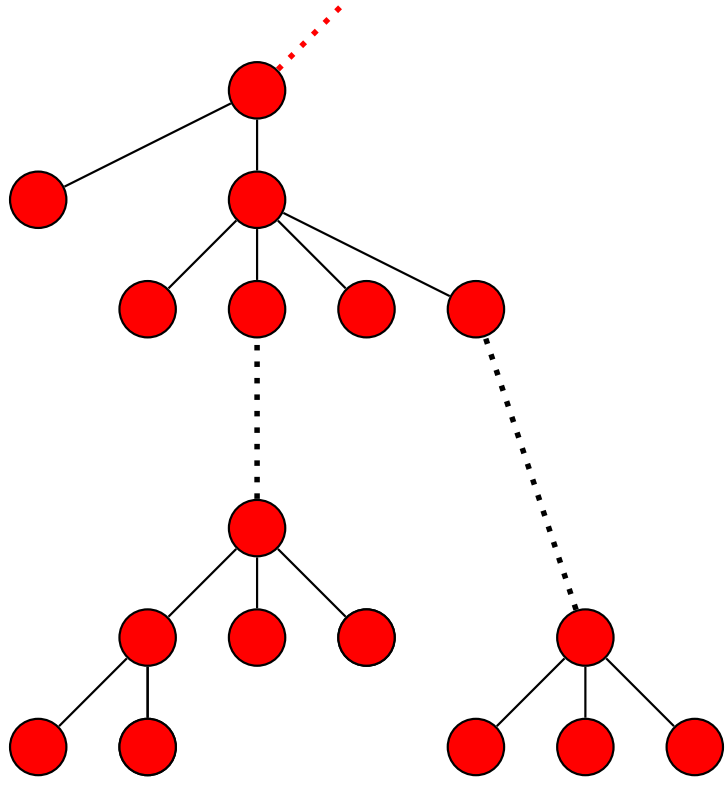


. . .



Some time later . . .





Well founded relation in Coq

A relation is well founded if all elements are accessible.

```
Inductive Acc (A:Type) (R:A->A->Prop) (x:A) : Prop :=
  Acc_intro :
    (forall y : A, R y x -> Acc R y) -> Acc R x.

Definition well_founded (A:Type) (R:A->A->Prop) :=
  forall a, Acc R a.
```

It is possible to define functions by recursion on the accessibility proof of an element (`Function` is based on this)

Proving that some relation is well-founded

Coq's Standard Library provides us with some useful examples of well-founded relations :

- The predicate `lt` over `nat` (but you can use measure instead)
- The predicate `Zwf` c, which is the restriction of $<$ to the interval $[c, \infty[$ of $\mathbb{Z}$.

More example : log10

```
Function log10 (n : Z) {wf (Zwf 1) n} : Z :=
  if Zlt_bool n 10 then 0 else 1 + log10 (n / 10).
Proof.
  (* first goal *)
  intros n Hleb.
  unfold Zwf.
  generalize (Zlt_cases n 10) (Z_div_lt n 10);rewrite Hleb.
  omega.
  (* Second goal *)
  apply Zwf_well_founded.
Defined.
(* Compute log10 2. : you can wait for a answer ... *)
```

log10 can also be defined using measure

```
Function log10 (n : Z) {measure Zabs_nat n} : Z :=
  if Zlt_bool n 10 then 0 else 1 + log10 (n / 10).
Proof.
  (* first goal *)
  intros n Hleb.
  unfold Zwf;generalize (Zlt_cases n 10); rewrite Hleb;intro
  apply Zabs_nat_lt.
  split.
  apply Z_div_pos;omega.
  apply Zdiv_lt_upper_bound;omega.
Defined.
```

Proving properties on recursive functions

Usually a property on a recursive function can be proved using an induction on the recursive argument.

Sometime we need some generalization before starting the induction (see the example `div2_le`).

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 x :=
  match x with 0 => 1 | S p => x * fact p end.

Functional Scheme fact_ind := Induction for fact Sort Prop.

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 div2t2le : forall x, div2 x * 2 <= x.
intros x; functional induction div2 x.
3 subgoals
  ============================
   0 * 2 <= 0

subgoal 2 is:
 0 * 2 <= 1
subgoal 3 is:
 S (div2 p) * 2 <= S (S p)
```

Proof techniques for mutual fixpoints

Properties on mutual fixpoints should be generally proved simultaneously.
Example:

```
forall n, even n = true -> exists p, n = 2*p
forall n, odd n = true -> exists p, n = 2*p + 1
```

Solution 1: prove both in one shot

```
Lemma even_odd_ok : forall n,
  (even n = true -> exists p, n = 2*p) /\
  (odd n = true -> exists p, n = 2*p + 1).
```

Use the lemma to prove the two disjoint lemmas:

```
Lemma even_ok :
  forall n, even n = true -> exists p, n = 2*p.

Lemma odd_ok :
  forall n, odd n = true -> exists p, n = 2*p + 1.
```

Solution 2: use mutual lemmas

```
Lemma even_ok :
  forall n, even n = true -> exists p, n = 2*p
with odd_ok :
  forall n, odd n = true -> exists p, n = 2*p + 1.
```

This a way to define a mutual fixpoint in proof mode

Prove for function defined using `Function`

Among the few lemmas that are generated by `Function`, the lemma `log10_equation` has the following statement, which expresses the intention of the original definition :

```
log10_equation
     : forall n : Z,
       log10 n = (if Zlt_bool n 10
                  then 0
                  else 1 + log10 (n / 10))
```

```
Goal log10 103=2.
repeat (rewrite log10_equation;simpl).
```

```
Goal log10 103=2.
repeat (rewrite log10_equation;simpl).
```
1 subgoal

$$==========$$

$$2 = 2$$

```
trivial.
Qed.
```

Functional Scheme

A functional scheme for function defined using `Function` is also automatically generated (no need to use `Functional Scheme`)

```
Check Merge_ind.
```

So, you can use `functional induction ...`

Proofs about uncurrified functions

```
Lemma Merge_count : forall x l1 l2,
  count x (Merge (l1, l2)) = count x l1 + count x l2.
```

- 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 (see the proofs)
  - use projectors
  - add equalities

Generating its own induction principle

Sometime, the generated induction principle is not what you need.

```
Inductive tree (A:Type) :=
 | Node : A -> list (tree A) -> tree A.

Check tree_ind.

tree_ind
  : forall (A : Type) (P : tree A -> Prop),
       (forall (a : A) (l : list (tree A)), P (Node A a l)) ->
       forall t : tree A, P t
```

You have to build your own induction principle

```
my_tree_ind : forall (A : Type)
  (P : tree A -> Prop) (Pl : list (tree A) -> Prop),
  (forall a l, Pl l -> P (Node _ a l)) ->
  Pl nil ->
  (forall t l, P t -> Pl l -> Pl (t :: l)) ->
  forall t, P t
```

The proof is in the file (`my_tree_ind`)