On General Recursion

Pierre Castéran

Paris, June 2010

When structural recursion is not enough : some examples

We present some simple case studies where the most natural recursion scheme does not fill the structural recursion constraint :

- Discrete logarithm (base 10),
- Euclidean division,
- Merging sorted lists,
- Binary search.

Computing the discrete logarithm (base 10)

Problem : Defining some function $\log 10 : \mathrm{Z} \rightarrow \mathrm{Z}$, satisfying :

$$\forall n, p : \mathrm{Z}, 0 \leq p \Rightarrow 10^{p} \leq n < 10^{p+1} \Rightarrow log10(n) = p$$

A first attempt could be :

```
Fixpoint log10 (n : Z) : Z :=
  if Zlt_bool n 10
  then 0
  else 1 + log10 (n / 10).
```

```
Fixpoint log10 (n : Z) : Z :=
  if Zlt_bool n 10
  then 0
  else 1 + log10 (n / 10).
```

```
Error:
Recursive definition of log10 is ill-formed.
In environment
log10 : Z -> Z
n : Z
Recursive call to log10 has principal argument equal to ”n / 10”
instead of a subterm of n.
```

Let us consider for instance the computation of $\log 10(253)$.

- This number is written `11111010` in binary form and the corresponding term is `Zpos(xO(xI(xO(xI(xI(xI(xI(xH))))))))`.
- The next recursive call is $\log 10(25)$; 25's binary representation is `11001`, and the associated *Coq* term is In *Coq*, this number is `Zpos(xI(xO(xO(xI(xH)))))`.
- Clearly, the subterm constraint is not satisfied by this computation.

Euclidean division

This example is very similar to `log10`. If we want to compute the euclidean division of *a* by *b* through successive subtractions, we don't respect the subterm condition.
Example : division of 100 by 27.

```
(100,27)
    (73,27)
       (46,27)
          (19,27)
          (0,19)
       (1,19)
    (2,19)
(3,19)
```

Merging two sorted lists

`merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =`

Merging two sorted lists

```
merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge(3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
```

Merging two sorted lists

```
merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge(3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: merge(3 :: 5 :: nil, 2 :: 4 :: 8 :: 34 :: nil) =
```

Merging two sorted lists

```
merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge(3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: merge(3 :: 5 :: nil, 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: merge(3 :: 5 :: nil, 4 :: 8 :: 34 :: nil) =
```

Merging two sorted lists

```
merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge(3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: merge(3 :: 5 :: nil, 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: merge(3 :: 5 :: nil, 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: merge(5 :: nil, 4 :: 8 :: 34 :: nil) =
```

Merging two sorted lists

$$
\begin{aligned}
&\mathtt{merge}(1 :: 3 :: 5 :: \mathtt{nil},\ 2 :: 2 :: 4 :: 8 :: 34 :: \mathtt{nil}) \ = \\
&1 :: \mathtt{merge}(3 :: 5 :: \mathtt{nil},\ 2 :: 2 :: 4 :: 8 :: 34 :: \mathtt{nil}) \ = \\
&1 :: 2 :: \mathtt{merge}(3 :: 5 :: \mathtt{nil},\ 2 :: 4 :: 8 :: 34 :: \mathtt{nil}) \ = \\
&1 :: 2 :: 2 :: \mathtt{merge}(3 :: 5 :: \mathtt{nil},\ 4 :: 8 :: 34 :: \mathtt{nil}) \ = \\
&1 :: 2 :: 2 :: 3 :: \mathtt{merge}(5 :: \mathtt{nil},\ 4 :: 8 :: 34 :: \mathtt{nil}) \ = \\
&1 :: 2 :: 2 :: 3 :: 4 :: \mathtt{merge}(5 :: \mathtt{nil},\ 8 :: 34 :: \mathtt{nil}) \ =
\end{aligned}
$$

Merging two sorted lists

```
merge(1 :: 3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge(3 :: 5 :: nil, 2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: merge(3 :: 5 :: nil, 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: merge(3 :: 5 :: nil, 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: merge(5 :: nil, 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: 4 :: merge(5 :: nil, 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: 4 :: 5 :: merge(nil, 8 :: 34 :: nil) =
```

Merging two sorted lists

$$
\begin{aligned}
&\texttt{merge}(1 :: 3 :: 5 :: \texttt{nil},\ 2 :: 2 :: 4 :: 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: \texttt{merge}(3 :: 5 :: \texttt{nil},\ 2 :: 2 :: 4 :: 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: \texttt{merge}(3 :: 5 :: \texttt{nil},\ 2 :: 4 :: 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: 2 :: \texttt{merge}(3 :: 5 :: \texttt{nil},\ 4 :: 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: 2 :: 3 :: \texttt{merge}(5 :: \texttt{nil},\ 4 :: 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: 2 :: 3 :: 4 :: \texttt{merge}(5 :: \texttt{nil},\ 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: 2 :: 3 :: 4 :: 5 :: \texttt{merge}(\texttt{nil},\ 8 :: 34 :: \texttt{nil}) \ =\\
&1 :: 2 :: 2 :: 3 :: 4 :: 5 :: 8 :: 34 :: \texttt{nil}
\end{aligned}
$$

Binary search

Let a be a sorted array, for instance :

i	1	2	3	4	5	6	7	8	9	10	11	12
$a(i)$	-10	-10	2	5	8	8	17	18	29	30	30	42

We look for instance for some index i such that $a(i) = 7$.

Looking for 7 in *a* between the indexes 1 and 12 (12 cells) amounts to look for 4 between the indexes 1 and 5 (5 cells), then between 4 and 5 (2 cells), etc.

1	2	3	4	5	6	7	8	9	10	11	12
-10	-10	2	5	8	8	17	18	29	30	30	42

Looking for 7 in a between the indexes 1 and 12 (12 cells) amounts to look for 4 between the indexes 1 and 5 (5 cells), then between 4 and 5 (2 cells), etc.

1	2	3	4	5	6	7	8	9	10	11	12
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42

Looking for 7 in *a* between the indexes 1 and 12 (12 cells) amounts to look for 4 between the indexes 1 and 5 (5 cells), then between 4 and 5 (2 cells), etc.

1	2	3	4	5	6	7	8	9	10	11	12
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42

Looking for 7 in *a* between the indexes 1 and 12 (12 cells) amounts to look for 4 between the indexes 1 and 5 (5 cells), then between 4 and 5 (2 cells), etc.

1	2	3	4	5	6	7	8	9	10	11	12
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42

Looking for 7 in *a* between the indexes 1 and 12 (12 cells) amounts to look for 4 between the indexes 1 and 5 (5 cells), then between 4 and 5 (2 cells), etc.

1	2	3	4	5	6	7	8	9	10	11	12
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42
−10	−10	2	5	8	8	17	18	29	30	30	42

Unfortunately, neither the number of cells nor the bounds give us a *structurally* decreasing argument.

Bounding the number of calls to a recursive function

It is sometimes possible to bound the number of calls to a recursive function. In this case, one can use this information for building a well-formed structural recursion.

For instance, when merging two lists u and v, the natural number $l = |u| + |v|$ decreases by 1 at each recursive call.

```
merge(1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
merge_aux 8 (1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
```

```
merge(1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
merge_aux 8 (1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge_aux 7 (3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
```

```
merge(1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
merge_aux 8 (1 :: 3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: merge_aux 7 (3 :: 5 :: nil) (2 :: 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: merge_aux 6 (3 :: 5 :: nil) ( 2 :: 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: merge_aux 5 (3 :: 5 :: nil) (4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: merge_aux 4 (5 :: nil, 4 :: 8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: 4 :: merge_aux 3 (5 :: nil) (8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: 4 :: 5 :: merge_aux 2 nil (8 :: 34 :: nil) =
1 :: 2 :: 2 :: 3 :: 4 :: 5 :: 8 :: 34 :: nil
```

```
 Function merge_aux (n : nat) (u v : list Z) {struct n}
 : list Z  :=
 match u,v,n with 0%nat, _, _ => nil
                | S _, u, nil => u
                | S _, nil, v => v
                | S p, a::u’,  b::v’  =>
                   if Zle_bool a b
                   then a::(merge_aux p u’ v )
                   else b::(merge_aux p u v’)
 end.

Definition merge u v :=
          merge_aux (length u + length v) u v .
```

A look at the extracted code

```
let rec merge_aux n u v =
  match n with
    | O -> Nil
    | S p ->
        (match u,v  with
            | Nil, Nil -> u
            | Nil,Cons (z0, l) -> v
            | Cons (a, u'), Nil -> u
            | Cons (a, u'),  Cons (b, v') ->
              if zle_bool a b
              then Cons (a, (merge_aux p u' v))
              else Cons (b, (merge_aux p u v')))))

let merge u v =  merge_aux (length u + length v) u v
```

Main drawbacks of this solution

- More computations than needed :
    1. Computation of the lists' length
    2. merging the lists

  This computation appears also in the extracted program.
- a correctness proof of `merge` must include a proof that the numeric argument given to `merge_aux` is large enough.

We shall now present some techniques for avoiding this extra work as much as possible.

Using Measures over Natural Numbers

We present a simple technique that allows the user to write recursive functions with less constraints than “pure” structural recursion. Furthermore, termination arguments are erased during extraction.

A first example

```
Require Import Recdef.

(* Zabs_nat : Z → nat *)

Function log10 (n : Z) {measure Zabs_nat n}: Z :=
  if Zlt_bool n 10
  then 0
  else 1 + log10 (n / 10).
```

We have to prove that the measure associated with the argument `n` strictly decreases along recursive calls.

```
1 subgoal

=========================
forall n : Z,
Zlt_bool n 10 = false -> (Zabs_nat (n / 10) < Zabs_nat n)%nat
```

The library `Recdef` allows *Coq* to accept this definition, once this goal (called a *proof obligation*) is solved.

Merge, with measures

```
Definition plus_length (u_v : list Z * list Z):nat :=
  (length (fst u_v) +  length (snd u_v))%nat.

Function merge (u_v : list Z * list Z)
  {measure plus_length u_v} : list Z :=
 match u_v with
  (nil,v) => v
| (u,nil) => u
| ((a::u')  as u,(b::v') as v) => if Zle_bool a b
                                  then a::(merge (u',v))
                                  else b::(merge (u,v'))
 end.
```

Binary search, with measures

```
Let m (p: Z * Z) : nat :=  Zabs_nat (snd p - fst p).

Function search (bounds : Z*Z )
{measure m   bounds} : option Z  :=
  let (from,to) := bounds in
  if Zle_bool from to
  then  let m := middle from to in
    if Zeq_bool x (a m)
    then Some m
    else if andb (Zle_bool from  (m-1)) (Zlt_bool x (a m))
       then search  (from,  m - 1)
       else if Zle_bool (m +1) to
             then search  (m+1, to)
             else None
 else None.
```

A more complex example

```
(* Example : pairs 4 returns the list
 (4,4)::(4,3)::(4,2)::(4,1)::(3,3)::(3,2)::
  (3,1)::(2,2)::(2,1)::(1,1)::nil *)

Function pairs_aux (p:nat*nat) {measure ????}:=
match p with (0,_) => nil
            |(S i, S j) => (S i,S j)::pairs_aux (S i,j)
            |(S i, 0) => pairs_aux (i, i)
end.

Definition pairs (i:nat) := pairs_aux (i,i).
```

No simple (linear) measure can be given to `Function`!

Let's consider a measure of the form :

`fun p: nat*nat =>` α`*(fst p)+`β`*(snd p)`

The measure of (S *i*,O) must be greater than the measure of (*i*,*i*), the same with (S *i*, S *j*) and (S *i*, *j*),

Thus, we should have $\beta > 0$ and $\alpha > \beta \times i$ for any *i*, which is impossible.

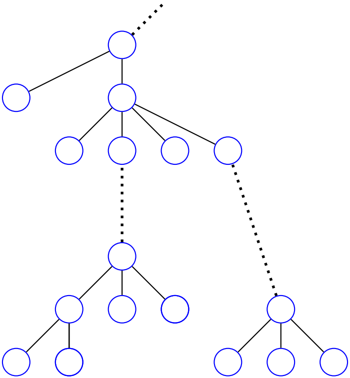
Solutions ?

Using a non-linear measure, like :

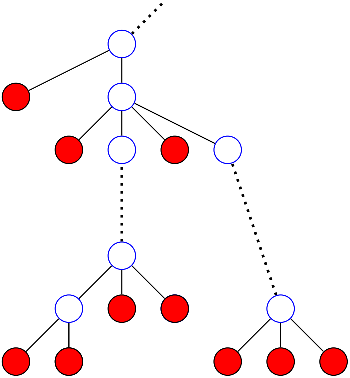
```
fun p : nat*nat => let (i,j) := p in i*(i+1)+ j
```

The proof must be done *manually*, because the automatic tactic `omega` doesn't work properly with multiplications.

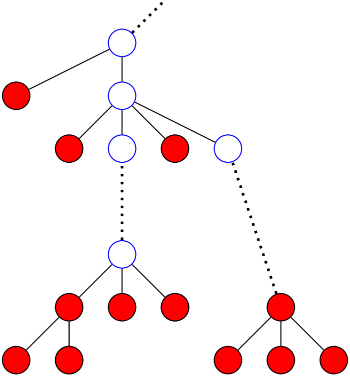
Well-founded Relations



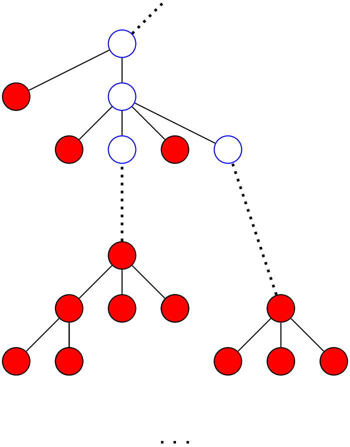
Dotted lines represent any number of elementary relationships

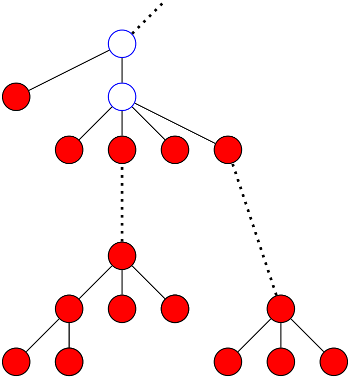


Minimal elements are *accessible*

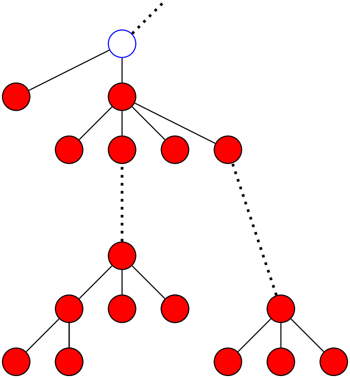


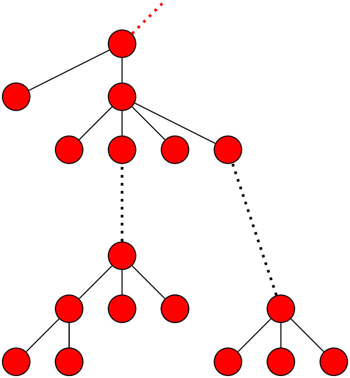
Elements whose all predecessors are accessible become accessible





Some time later . . .





Termination using well-founded relations

For proving that some recursive function f with main argument $a : A$ terminates, hence is acceptable by *Coq* :

1. Consider some well-founded relation R over A
2. Prove that for each recursive call $f\ b$, $b\ R\ a$ holds.

We have to use the `wf` option of `Function` :

```
Function f (x:A1) (a : A) {wf R a} : B :=
... f y1 b ...
```

Termination using well-founded relations

For proving that some recursive function f with main argument $a : A$ terminates, hence is acceptable by *Coq* :

1. Consider some well-founded relation R over A
2. Prove that for each recursive call $f\ b$, $b\,R\,a$ holds.

We have to use the `wf` option of `Function` :

```
Function f (x:A1) (a : A) {wf R a} : B :=
... f y1 b ...
```

This command generates two kinds of proof obligations :

- Proving the relations $b\,R\,a$, under the hypotheses associated to the context of each recursive call to f,
- Proving that R is truly well-founded.

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 $\mathbb{Z}$.

```
Function log10 (n : Z){wf  (Zwf 1)  n} : Z :=
  if Zlt_bool n 10
  then 0
  else 1 + log10 (n / 10).
Proof.
intros n teq;Zbool2Prop.
generalize (Z_div_lt n 10);intros;split;omega.
apply Zwf_well_founded.
Qed.
```

Using the Standard Library

The Standard Library provides the user with some useful theorems that allow to prove some relation is well-founded.

```
Require Import Inclusion.
Require Import Zwf.

Lemma half_wf : well_founded
                 (fun i j : Z =>  0 < i ∧ j = 2 * i).
Proof.
 apply wf_incl with (Zwf 0).
 (* prove that our relation is included in (Zwf 0) *)
 intros i j [H H0];split;auto with zarith.
 apply Zwf_well_founded.
Qed.
```

- Other modules (in the `Wellfounded` section of the Standard Library) contain similar lemmas. Their use is interesting, but requires some experience with the *Coq* system.
- It is possible to design tools for adapting these lemmas to the use of `Function`.

```
Require Import Measures.(* Not in Standard Library *)

Let measures := (@fst nat nat) ::
                (@snd nat nat) :: nil.

Function pairs_aux (p:nat*nat)
         {wf (measures_lt  measures)  p}
         : list (nat*nat):=
match p with (0,_) => nil
            |(S i, S j) => (S i, S j)::pairs_aux (S i, j)
            |(S i, 0) => pairs_aux (i, i)
end.
```

Testing the function

Once *Coq* has accepted your function, and before proving it is correct, it may be useful to do some simple tests :

```
Eval compute in log10 67.
```

Testing the function

Once *Coq* has accepted your function, and before proving it is correct, it may be useful to do some simple tests :

```
Eval compute in log10 67.
```

waiting for an answer . . .

In fact, `log10` is defined by a huge *Coq* term, which contains all the termination proof. Just try to type :

```
Goal log10 67 = 2.
Proof.
unfold log10, log10_terminate;simpl.
```

A goal of more than 450 lines !

Using the extraction facility

```
Extraction "log10.ml" log10.
```

Using the extraction facility

```
Extraction "log10.ml" log10.
```

The file log10.ml has been created by extraction.

The file log10.mli has been created by extraction.

Using the extraction facility

```
Extraction "log10.ml" log10.
The file log10.ml has been created by extraction.
The file log10.mli has been created by extraction.
```

```
let log_10 x = z_to_int (log10 (int_to_Z x));;

log_10 9999;;
```

Using the extraction facility

```
Extraction "log10.ml" log10.
The file log10.ml has been created by extraction.
The file log10.mli has been created by extraction.

let log_10 x = z_to_int (log10 (int_to_Z x));;

log_10 9999;;
- : int = 3
log_10 10000;;
```

Using the extraction facility

```
Extraction "log10.ml" log10.
The file log10.ml has been created by extraction.
The file log10.mli has been created by extraction.

let log_10 x = z_to_int (log10 (int_to_Z x));;

log_10 9999;;
- : int = 3
log_10 10000;;
- : int = 4
log_10 0;;
```

Using the extraction facility

```
Extraction "log10.ml" log10.
The file log10.ml has been created by extraction.
The file log10.mli has been created by extraction.

let log_10 x = z_to_int (log10 (int_to_Z x));;

log_10 9999;;
- : int = 3
log_10 10000;;
- : int = 4
log_10 0;;
- : int = 0
```

Proving equalities in *Coq*

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.
```

Correctness Proofs

`log10`'s correctness is expressed by the following statement, which relates the argument n and the result log10 n :

```
Lemma log10_OK :
forall n p, 0 ≤ p ->
           10 ^ p ≤ n < 10^(p+1)->
           log10 n = p.
```

```
intro n.
1 subgoal
  n : Z
  ============================
   forall p : Z, 0 ≤ p ->
    10 ^ p ≤ n < 10 ^ (p + 1) ->
    log10 n = p
functional induction (log10 n).
```

A first subgoal is generated from the structure of the function definition :

```
Function log10 (n : Z) ...
  if Zlt_bool n 10
  then 0  ...
```

```
  n : Z
  e : Zlt_bool n 10 = true
  p : Z
  H : 0 ≤ p
  H0 : 10 ^ p ≤ n < 10 ^ (p + 1)
============================
   0 = p
```

Let us consider the `else` part of the function definition, which contains a recursive call.

```
Function log10 (n : Z)wf  (Zwf 1)  n : Z :=
  if Zlt_bool n 10
  ...
  else 1 + log10 (n / 10).
```

Coq generates a context assuming the recursive calls correctness, and a goal for proving the correctness of the computed result.

```
 e : Zlt_bool n 10 = false
IHz : forall p : Z,
      0 ≤ p ->
      10 ^ p ≤ n / 10 < 10 ^ (p + 1) ->
      log10 (n / 10) = p
H : 0 ≤ p
H0 : 10 ^ p ≤ n < 10 ^ (p + 1)
============================
 1 + log10 (n / 10) = p
```

For solving this goal, we first assert that $0 < p$ (from `e` and `H0`), then apply `IHz` to $p - 1$.

```
  e : Zlt_bool n 10 = false
IHz : forall p : Z,
      0 ≤ p ->
      10 ^ p ≤ n / 10 < 10 ^ (p + 1) ->
      log10 (n / 10) = p
H : 0 ≤ p
H0 : 10 ^ p ≤ n < 10 ^ (p + 1)
============================
 1 + log10 (n / 10) = p
```

For solving this goal, we first assert that $0 < p$ (from `e` and `H0`), then apply `IHz` to $p - 1$. *The actual proof uses properties of exponentiation, the arithmetic solver `omega`, and conversions between `Zlt_bool` and $<$.*

More examples

In this section, we present roughly some techniques we have used in our correctness proofs. Full proofs are either left as exercises or can be downloaded from the school's page.

Require Import Recdef.

```
Definition plus_length (u_v : list Z * list Z):nat :=
   length (fst u_v) + length (snd u_v).

Function merge (u_v : list Z * list Z)
 {measure plus_length u_v} : list Z :=
 match u_v with
| (nil,v) => v
| (u,nil) => u
| ((a::u')  as u,(b::v') as v) =>
    if Zle_bool a b
    then a::(merge (u',v))
    else b::(merge (u,v'))
 end.
```

We want to prove that, if *u* and *v* are sorted, then `merge (u,v)` is sorted too.

```
Inductive sorted : list Z -> Prop :=
| sorted_nil : sorted nil
| sorted_single : forall a, sorted (a::nil)
| sorted_2 : forall a b l, a ≤ b -> sorted (b::l) ->
                sorted (a::b::l).

Hint Constructors sorted.

Lemma sorted_merge_0: forall u_v, sorted (fst u_v) ->
                             sorted (snd u_v) ->
                             sorted (merge u_v).
```

```
intro u_v; functional induction (merge u_v) ;simpl.
```

The tactic call functional induction (merge u_v) considers 4 situations, each one corresponding to the possible results (in blue). When needed, an induction hypothesis is generated for the recursive calls (in red).

The first subgoal corresponds to the case where u is empty :

```
 (* match u_v with
    | (nil,v) => v
    ...
*)
```

```
 v : list Z
  ============================
   sorted nil -> sorted v -> sorted v
```

The second subgoal is quite the same (up to symmetry).

Let us consider the third subgoal :

```
(*...
| ((a::u')  as u,(b::v') as v) =>
 if Zle_bool a b then a::(merge (u',v))
...*)
```

e0 : Zle_bool a b = true

IHl : sorted u' -> sorted (b :: v') ->

sorted (merge (u', b :: v'))

H : sorted (a :: u')

H0 : sorted (b :: v')

============================

sorted (a :: merge (u', b :: v'))

```
e0 : Zle_bool a b = true
 IH1 : sorted u' -> sorted (b :: v') ->
      sorted (merge (u', b :: v'))
 H : sorted (a :: u')
 H0 : sorted (b :: v')
 ============================
  sorted (a :: merge (u', b :: v'))
```

Inversion on `H` and `H0` leads to consider some cases :

- $u' = nil$ or $u' = a0 :: w$ (with $a \leq a0$)
- $v' = nil$ or $v' = b0 :: v''$ (with $b \leq b0$)
- comparison of $a0$ with $b0$

For instance, in the following situation :

```
e0 : Zle_bool a b = true
IH1 : sorted (a0 :: w) ->
      sorted (b :: nil) ->
      sorted (merge (a0::w, b :: nil))
H : sorted (a :: a0 :: w)
H2 : a ≤ a0
H3 : sorted (a0 :: w)
H4 : sorted (b :: nil)
 ============================
 sorted (a :: merge (a0 :: w, b :: nil))
```

Using `merge_equation` (twice), then comparing *a*0 and *b* helps us to solve the goal.

```
H2 : a ≤ a0
H3 : sorted (a0::w)
IHl : sorted (a0::w) -> sorted (b::nil) ->
      sorted (a0::merge (w,b::nil))
eg : Zle_bool a0 b = true
==================================
 sorted (a::a0::merge (w, b::nil))
auto.
```

Binary Search

```
Function search (bounds : Z*Z ){wf R bounds} :
option Z  :=
  let (from,to) := bounds in
  if Zle_bool from to
  then
   let m := middle from to in
     if Zeq_bool x (a m)
     then Some m
     else if andb (Zle_bool from (m-1)) (Zlt_bool x (a m))
       then search (from, m-1)
       else if Zle_bool (m +1) to
             then search (m+1, to)
              else None
 else None.
```

We want to prove, for instance, that if the array *a* is sorted from *from* to *to*, and `search` $a\ x\ (from, to)$ returns `None` then *x* doesn't occur in *a*. More precisely :

```
  forall from to : Z,
   from ≤ to ->
   sorted from to ->
   search (from, to) = None ->
 forall k : Z, from ≤ k ≤ to -> a k <> x
```

As for `merge`, the proof has the following main steps :
Conversion of the statement into the following form :

```
H' : from ≤ to
p := (from, to) : Z * Z
H : sorted from to
H0 : search p = None
k : Z
H1 : from ≤ k ≤ to
H2 : a k = x
============================
 False
```

Then do functional induction (search p).

We have to solve some goals like the following one (where $from < m \leq to$ and $x < a(m)$)

```
H' : from ≤ to
H : sorted from to
m := middle from to : Z
H0 : search (from, m - 1) = None
H1 : from ≤ k ≤ to
IHo : from ≤ m - 1 -> sorted from (m - 1) ->
      search (from, m - 1) = None ->
      from ≤ k ≤ m - 1 -> a k <> x
H3 : from ≤ m - 1
H4 : x < a m
m_1 : from ≤ m ≤ to
============================
a k <> x
```

Conclusion and exercises

- Complete the example on `merge`.
- **(difficult)** Prove that `Zlt` is not well-founded.
  Hint :
  1. Assume `Zlt` is well-founded,
  2. Define the following function :

     ```
     Function loop (z:Z){wf Zlt z} : Z:=  loop(z-1).
     ```

  3. Prove that for all z, `loop z = loop z + 1`
  4. Get a contradiction from all that.