Récurseurs et condition de garde

Thierry Martinez

vendredi 22 mars 2024

Comment écrire des programmes sur des structures inductives ?

```
Inductive nat :=
| O : nat
| S : nat -> nat.
```

De cette définition, Coq dérive quatre principe d'inductions : `nat_rect`, `nat_ind`, `nat_rec`, `nat_sind`.

et permet de définir des fonctions récursives par point fixe :

```
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => S (add m' n)
  end.
```

définition à laquelle Coq répond

```
add is recursively defined (guarded on 1st argument)
```

Quel est le lien entre tout ça, et comment la terminaison est-elle garantie ?

Le récurseur nat_rec

```
Print nat_rec.
nat_rec =
fun P : nat -> Set => nat_rect P
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Paraphrasons : pour toute famille de type indexée $(\mathtt{P\ n})_n$,

- étant donné un élément de type P_0 (cas de base) et
- étant donnée une fonction qui calcule à partir d'un élément de type P_n un élément de type $P_{S(n)}$,

alors on en déduit une fonction qui calcule pour tout entier n, un élément P_n.

Application : add peut être défini par récurseur, en prenant $\forall \mathtt{n}, \mathtt{P\ n} = \mathtt{nat}$.

```
Definition add' (m n : nat) :=
  nat_rec (fun _ => nat) n (fun _ acc => S acc) m.
```

Utilisation de nat_rec avec un type dépendant

```
Inductive vect (A : Set) : nat -> Set :=
| nil : vect A O
| cons : forall n, A -> vect A n -> vect A (S n).

Fixpoint fill (A : Set) (x : A) (n : nat) : vect A n :=
  match n with
  | O => nil A
  | S n' => cons A n' x (fill A x n')
  end.
```

Définition de fill avec nat_rec ?

Utilisation de nat_rec avec un type dépendant

```
Inductive vect (A : Set) : nat -> Set :=
| nil : vect A O
| cons : forall n, A -> vect A n -> vect A (S n).

Fixpoint fill (A : Set) (x : A) (n : nat) : vect A n :=
  match n with
  | O => nil A
  | S n' => cons A n' x (fill A x n')
  end.
```

Définition de fill avec nat_rec ?

```
Definition fill' (A: Set) (x: A) (n: nat) : vect A n :=
  nat_rec (fun n => vect A n) (nil A)
    (fun n' acc => cons A n' x acc) n.
```

Définition de nat_rec par point fixe

```
Fixpoint nat_rec' (P : nat -> Set) (P0 : P O)
  (PH : forall n, P n -> P (S n)) (n : nat) : P n :=
```

Définition de nat_rec par point fixe

```
Fixpoint nat_rec' (P : nat -> Set) (P0 : P 0)
  (PH : forall n, P n -> P (S n)) (n : nat) : P n :=
  match n with
  | 0 => P0
  | S n' => PH n' (nat_rec' P P0 PH n')
  end.
```

match plus économe que nat_rec

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

Définition de pred avec nat_rec ?

```
Definition pred' (n : nat) : nat :=
```

match plus économe que nat_rec

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

Définition de pred avec nat_rec ?

```
Definition pred' (n : nat) : nat :=
  fst (nat_rec (fun _ => (nat * nat)%type) (O, O)
    (fun n' '(p, q) => (q, S q)) n).
```

match sur deux structures en parallèle

```
Fixpoint sub (m n : nat) : nat :=
  match m, n with
  | O, _ => O
  | _, O => m
  | S m', S n' => S (sub m' n')
  end.
```

Définition de sub avec nat_rec ?

```
Definition sub' (m n : nat) : nat :=
```

`match` sur deux structures en parallèle

```
Fixpoint sub (m n : nat) : nat :=
  match m, n with
  | O, _ => O
  | _, O => m
  | S m', S n' => S (sub m' n')
  end.
```

Définition de `sub` avec `nat_rec` ?

```
Definition sub' (m n : nat) : nat :=
  nat_rec (fun _ => nat) m (fun _ acc => pred acc) n.
```

Complexité de `sub'` : $\mathbf{O}(\mathtt{m} \times \mathtt{n})$!

Le récurseur vect_rec

```
Inductive vect (A : Set) : nat -> Set :=
| nil : vect A O
| cons : forall n, A -> vect A n -> vect A (S n).
```

```
Print vect_rec.
vect_rec =
fun (A : Set) (P : forall n : nat, vect A n -> Set) =>
  vect_rect A P
: forall (A : Set) (P : forall n : nat, vect A n -> Set),
  P O (nil A) ->
  (forall (n : nat) (a : A) (v : vect A n),
   P n v -> P (S n) (cons A n a v)) ->
  forall (n : nat) (v : vect A n), P n v
```

map pour vect

```
Fixpoint map (A B : Set) (n : nat) (f : A -> B)
  (v : vect A n) : vect B n :=
```

map pour vect

```
Fixpoint map (A B : Set) (n : nat) (f : A -> B)
  (v : vect A n) : vect B n :=
  match v with
  | nil _ => nil B
  | cons _ n' hd tl => cons B n' (f hd) (map A B n' f tl)
  end.
```

Définition de map avec vect_rec ?

```
Definition map' (A B : Set) (n : nat) (f : A -> B)
  (v : vect A n) : vect B n :=
```

map pour vect

```
Fixpoint map (A B : Set) (n : nat) (f : A -> B)
  (v : vect A n) : vect B n :=
  match v with
  | nil _ => nil B
  | cons _ n' hd tl => cons B n' (f hd) (map A B n' f tl)
  end.
```

Définition de map avec vect_rec ?

```
Definition map' (A B : Set) (n : nat) (f : A -> B)
  (v : vect A n) : vect B n :=
  vect_rec A (fun n' _ => vect B n')
    (nil B)
    (fun n' hd _ acc => cons B n' (f hd) acc) n v.
```

Le récurseur tree_rec

```
Inductive tree (A : Set) :=
| leaf
| node (lhs : tree A) (label : A) (rhs : tree A).

Print tree_rec.
```

Le récurseur tree_rec

```
Inductive tree (A : Set) :=
| leaf
| node (lhs : tree A) (label : A) (rhs : tree A).
```

```
Print tree_rec.
```

```
tree_rec =
fun (A : Set) (P : tree A -> Set) => tree_rect A P
     : forall (A : Set) (P : tree A -> Set),
       P (leaf A) ->
       (forall lhs : tree A,
        P lhs ->
        forall (label : A)
        (rhs : tree A), P rhs ->
        P (node A lhs label rhs)) ->
       forall t : tree A, P t
```

height pour tree

```
Fixpoint height (A : Set) (t : tree A) : nat :=
  match t with
  | leaf _ => O
  | node  _ lhs _ rhs =>
      S (max (height A lhs) (height A rhs))
  end.
```

Définition de `height` avec `tree_rec` ?

```
Definition height' (A : Set) (t : tree A) : nat :=
```

height pour tree

```
Fixpoint height (A : Set) (t : tree A) : nat :=
  match t with
  | leaf _ => 0
  | node  _ lhs _ rhs =>
      S (max (height A lhs) (height A rhs))
  end.
```

Définition de height avec tree_rec ?

```
Definition height' (A : Set) (t : tree A) : nat :=
  tree_rec A (fun _ => nat) 0
    (fun _ hl _ _ hr => S (max hl hr)) t.
```

Condition de garde : un argument décroit strictement

```
Fixpoint height (A : Set) (t : tree A) {struct t}: nat :=
  match t with
  | leaf _ => 0
  | node  _ lhs _ rhs =>
      S (max (height A lhs) (height A rhs))
  end.
```

`lhs` et `rhs` sont obtenus par filtrage de `t` à travers le constructeur `node` **en position récursive** (`tree` apparaît en argument du constructeur) :

```
Inductive tree (A : Set) :=
| leaf
| node (lhs : tree A) (label : A) (rhs : tree A).
```

ceci garantit `lhs` $<$ `t` et `rhs` $<$ `t` pour l'ordre structurel.

Condition de garde : stabilité par application (et abstraction)

```
Inductive tree' (A : Set) :=
| leaf'
| node' (label : A) (children : bool -> tree' A).

Print tree'_rec.

Fixpoint height' (A : Set) (t : tree' A) : nat :=
  match t with
  | leaf' _ => O
  | node'  _ _ children =>
      S (max (height' A (children false))
             (height' A (children true)))
  end.
```

si $u < t$ alors $u\ e < t$ (on a aussi `fun x => u` $< t$).

Condition de garde : seuls les sous-termes en position récursive sont pris en compte

Par imprédicativité, on peut avoir des sous-termes aussi grands que le terme lui-même.

```
Inductive I : Prop := C (f: forall A:Prop, A->A).
Fail Fixpoint F (x:I) : False :=
  match x with
  | C f => F (f I x)
  end.
```

Condition de garde : stabilité par filtrage

```
Inductive direction := Left | Right.

Fixpoint path_height (t : tree direction) : nat :=
  match t with
  | leaf _ => O
  | node _ lhs label rhs =>
      S (path_height
          (match label with
           | Left => lhs
           | Right => rhs
           end))
  end.
```

Si toutes les branches sont telles que `b` $<$ `t` alors
`match _ with _ => b end` $<$ `t`.

Terminaison et condition de garde

Théorème

Pour toute définition par point fixe qui vérifie la condition de garde, il existe une définition par récurseur qui lui est extentionnellement équivalente.

Eduarde Giménez, Codifying guarded definitions with recursive schemes, TYPES 1994 : Types for Proofs and Programs

Récurseurs pour les inductifs dans Prop

Un terme de Prop ne peut être éliminé dans Set : pas de récurseur rec (ni rect a fortiori), on a seulement ind (élimination dans Prop) et sind (élimination dans SProp).
D'autre part, les termes récursifs ne portent pas d'information (*proof irrelevance*) : ils ne sont pas passés en argument du prédicat d'élimination.

```
Inductive even : nat -> Prop :=
| even_0: even 0
| even_SS: forall n, even n -> even (S (S n)).
Print even_ind.
even_ind =
fun (P : nat -> Prop) (f : P 0)
  (f0 : forall n : nat, even n -> P n -> P (S (S n))) =>
[...] : forall P : nat -> Prop,
        P 0 ->
        (forall n : nat, even n -> P n -> P (S (S n))) ->
        forall n : nat, even n -> P n
```

Exemple d'inductif dans Prop : l'égalité

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq x x.

Print eq_ind.
```

Exemple d'inductif dans Prop : l'égalité

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq x x.

Print eq_ind.
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (f : P x) (a : A)
  (e : eq x a) =>
match e in eq _ a0 return P a0 with
| eq_refl => f
end
     : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall a : A, eq x a -> P a
```

Exemple d'élimination dans Prop

```
Fixpoint add_r (n : nat) : add n O = n :=
  match n with
  | O => eq_refl
  | S n' =>
```

Exemple d'élimination dans Prop

```
Fixpoint add_r (n : nat) : add n O = n :=
  match n with
  | O => eq_refl
  | S n' =>
      match add_r n' in _ = m
      return add (S n') O = S m with
      | eq_refl => eq_refl
      end
  end.
```

Définition avec nat_ind et eq_ind ?

Exemple d'élimination dans Prop

```
Fixpoint add_r (n : nat) : add n O = n :=
  match n with
  | O => eq_refl
  | S n' =>
      match add_r n' in _ = m
      return add (S n') O = S m with
      | eq_refl => eq_refl
      end
  end.
```

Définition avec nat_ind et eq_ind ?

```
Definition add_r' (n : nat) : add n O = n :=
  nat_ind (fun n' => add n' O = n') eq_refl
    (fun n' IH =>
      eq_ind (add n' O) (fun m => add (S n') O = S m)
        eq_refl n' IH) n.
```

Ordres bien fondés

```
Import Wf.

Print Acc.
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.

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

Induction bien fondées

```
Check well_founded_induction.
well_founded_induction
     : forall (A : Type) (R : A -> A -> Prop),
       well_founded R ->
       forall P : A -> Set,
       (forall x : A,
         (forall y : A, R y x -> P y) -> P x) ->
       forall a : A, P a
```

lt est bien fondé

```
Require Import PeanoNat.

Theorem lt_wf: well_founded lt.
Proof.
```

Induction bien fondée sur `lt`

```
Require Import Compare_dec.
Import Wf.
Check zerop.
zerop : forall n : nat, {n = 0} + {0 < n}
Check Nat.lt_div2.
Nat.lt_div2 : forall n : nat, 0 < n -> Nat.div2 n < n

Definition log2_rec (n : nat)
  (IH : forall m, m < n -> nat) :=
```

Induction bien fondée sur `lt`

```
Require Import Compare_dec.
Import Wf.
Check zerop.
zerop : forall n : nat, {n = 0} + {0 < n}
Check Nat.lt_div2.
Nat.lt_div2 : forall n : nat, 0 < n -> Nat.div2 n < n

Definition log2_rec (n : nat)
  (IH : forall m, m < n -> nat) :=
  match zerop n with
  | left n_eq_0 => 0
  | right O_lt_n =>
    S (IH (Nat.div2 n) (Nat.lt_div2 n O_lt_n))
  end.

Definition log2 (n : nat) :=
```

Induction bien fondée sur lt

```
Require Import Compare_dec.
Import Wf.
Check zerop.
zerop : forall n : nat, {n = 0} + {0 < n}
Check Nat.lt_div2.
Nat.lt_div2 : forall n : nat, 0 < n -> Nat.div2 n < n

Definition log2_rec (n : nat)
  (IH : forall m, m < n -> nat) :=
  match zerop n with
  | left n_eq_0 => 0
  | right 0_lt_n =>
    S (IH (Nat.div2 n) (Nat.lt_div2 n 0_lt_n))
  end.

Definition log2 (n : nat) :=
  well_founded_induction lt_wf (fun _ => nat) log2_rec n.
```

Élimination des cas impossibles : filtrage sur les index

```
Definition tail A n (v : vect A (S n)) : vect A n :=
```

Élimination des cas impossibles : filtrage sur les index

```
Definition tail A n (v : vect A (S n)) : vect A n :=
  match v in vect _ m return
        match m with
        | O => unit
        | S n' => vect A n'
        end
  with
  | nil _ => tt
  | cons _ _n _hd tl => tl
  end.
```

Remarque : dans les cas simples, Coq élabore automatiquement le filtrage sur les cas impossibles

```
Definition tail A n (v : vect A (S n)) : vect A n :=
  match v with
  | cons _ _n _hd tl => tl
  end.
```

Filtrage dépendant : utilisation des coupures commutatives

```
Fixpoint map2 (A B C: Set) (f: A->B->C) n (u: vect A n)
  (v: vect B n): vect C n :=
```

Filtrage dépendant : utilisation des coupures commutatives

```
Fixpoint map2 (A B C: Set) (f: A->B->C) n (u: vect A n)
  (v: vect B n): vect C n :=
 match u in vect _ m return vect B m -> vect C m with
 | nil _ => fun _ => nil _
 | cons _ m' uh ut => fun (v' : vect B (S m')) =>
     match v' in vect _ m' return
       match m' with
       | O => unit
       | S m'' => vect A m'' -> vect C m'
       end with
     | nil _ => tt
     | cons _ m' vh vt => fun (ut' : vect A m') =>
         cons _ m' (f uh vh) (map2 A B C f m' ut' vt)
     end ut
 end v.
```

La condition de garde est généralisée pour passer à travers les coupures commutatives.