SU/FSI/master/info/stl/MU5IN554

Spécification et Vérification de Programmes

Novembre 2020

Pascal Manoury
pascal.manoury@lip6.fr

Fonctions récursives
Terminaison

Types et terminaison

Typage fort
Théorème fondamental

`f : nat -> nat` ssi pour tout `n:nat`, `(f n):nat`

`f` est *totalement définie* pour l'*évaluation symbolique*[1]
sur le domaine des expressions de type `nat`

C'est-à-dire : pour toute expression `e:nat`

il existe une expression `e':nat` *non réductible*[2] telle que
l'application `(f e)` se réduit en `e'`.

En conséquence :

pas de définition récursive de fonction
dont on ne sache garantir la terminaison.

---

1. réduction
2. pour laquelle il n'y a plus de règle d'évaluation applicable

Pourquoi

Soit `A:Type`, si on autorisait

```
Definition loop (n:nat) : A := (loop n).
```

On aurait, pour tout `A:Type`

```
loop : nat -> A
```

Or `False : Prop : Type`

On aurait donc :

- `loop : nat -> False`
- `(loop 0) : False`

C'est-à-dire une preuve de `False` sans hypothèse

D'où l'on pourrait tirer `F` et `not F`

Coq serait incohérent

Définition récursive

La clause `Fixpoint` soumet les définitions récursives à un *contrôle syntaxique* de terminaison.

Origine : les fonctions *primitives récursives*

1. `0` et `S` sont primitives récursives
2. fonctions constantes, projections
3. schéma (syntaxique) de définition récursive

$$\begin{cases} \texttt{(f 0 y)} & = & \texttt{(g y)} \\ \texttt{(f (S x) y)} & = & \texttt{(h x y (f x y))} \end{cases}$$

Bonne fondation

Pas de suite infinie de réduction

car

pas de suite infinie $(\mathtt{S}^n\ \mathtt{0})\ (\mathtt{S}^{n-1}\ \mathtt{0})\ \ldots$

$$
\begin{aligned}
&\mathtt{(f\ (S}^n\ \mathtt{0)\ y)}\\
&\quad = \mathtt{(h\ (S}^{n-1}\ \mathtt{0)\ y\ (f\ (S}^{n-1}\ \mathtt{0)\ y))}\\
&\qquad \vdots\\
&\quad = \mathtt{(h\ (S}^{n-1}\ \mathtt{0)\ y\ (h\ \ldots(h\ 0\ y\ (f\ 0\ y))\ldots))}\\
&\quad = \mathtt{(h\ (S}^{n-1}\ \mathtt{0)\ y\ (h\ \ldots(h\ 0\ y\ (g\ y))\ldots))}
\end{aligned}
$$

Le calcul (symbolique) *résoud* la récurrence

Schéma PR en Coq

```
Fixpoint pr (g:nat -> nat) (h:nat -> nat -> nat -> nat)
            (x y:nat) :=
  match x with
    0 => (g y)
  | (S x) => (h x y (pr g h x y))
  end.
```

Trivialement :

1. `(pr g h 0 y) = (g y)`
2. `(pr g h (S x) y) = (h x y (pr g h x y))`.

En posant : `Definition f := (pr g h).`
On retrouve :

- `(f 0 y) = (g y)`
- `(f (S x) y) = (f x y (f x y))`

L'addition est primitive récursive

```
Definition addpr :=
  let g x := x in
  let h x y r := (S r) in
    (pr g h).
```

On montre :

```
Lemma add_is_pr : forall (x y:nat),
    (addpr x y) = (x + y).
```

Par induction sur `x`.

La multiplication aussi

```
Definition mulpr :=
  let g x := 0 in
  let h x y r := y + r in
    (pr g h).

Lemma mul_is_pr : forall (x y:nat),
    (mulpr x y) = (x * y).
Proof.
  induction x.
  - trivial.
  - intro.
    unfold mulpr. simpl. rewrite IHx.
    trivial.
Qed.
```

Généralisation du schéma PR I

Autres types

Les listes

```
Fixpoint f {A:Type} (xs:(list A)) y :=
  match xs with
    nil => (g y)
  | x::xs => (h x xs y (f xs y))
  end.
```

Les arbres binaires

```
Fixpoint f {A:Type} (bt:(btree A) y :=
  match bt with
    Empty => (g y)
  | Node(bt1,x,bt2) =>
      (h bt1 x bt2 y (f bt1 y) (f bt2 y))
```

Tout type inductif : schéma PR $\approx$ récurrence structurelle

Au delà du schéma PR

Critère de terminaison

Théorie : ordre bien fondé

Pratique : syntaxe, *variable issue du filtrage*

```
Print pr.
pr =
fix pr (g : nat -> nat) (h : nat -> nat -> nat -> nat)
       (x y : nat) {struct x} : nat :=
  match x with
  | 0 => g y
  | S x0 => h x0 y (pr g h x0 y)
  end
  : (nat -> nat) -> (nat -> nat -> nat -> nat)
    -> nat -> nat -> nat
```

Terme de point fixe : une *variable de récurrence* (*variant*)

Généralisation du pas de récurrence

```
Fixpoint div2 (n:nat) : nat :=
  match n with
    0 => 0
  | 1 => 0
  | S (S n) => S (div2 n)
  end.

Fixpoint list_pairing {A:Type}
  (xs : (list A)) : (list (A * A)) :=
  match xs with
    nil => nil
  | (x::nil) => nil
  | (x1::(x2::xss)) =>
       ((x1,x2)::(list_pairing xss))
  end.
```

Deux pas à la fois

```
Fixpoint fib (n:nat) : nat :=
  match n with
    0 => 1
  | 1 => 1
  | (S (S n)) => (fib n) + (fib (S n))
  end.
```

Un sybillin message d'erreur

```
Error:
Recursive definition of fib is ill-formed.
[...]
Recursive call to fib has principal argument equal to
"S n1" instead of one of the following variables: "n0"
"n1".
[..]
```

(Trop) strict critère syntaxique

Variable issue du filtrage

Dans `(fib n) + (fib (S n))`, `(S n)` n'est pas une variable.

Expliciter les pas de filtrage :

```
Fixpoint fib (n:nat) : nat :=
  match n with
    0 => 1
  | (S n1) =>
      match n1 with
        0 => 1
      | (S n2) => (fib n2) + (fib n1)
      end
  end.
```

Double récurrence

```
Fixpoint shuffle {A:Type}
         (xs ys : (list A)) : (list A) :=
  match xs, ys with
    nil, _ => ys
  | _, nil => xs
  | (x::xs), (y::ys) => (x::(y::(shuffle xs ys)))
  end.
```

Apparente double récurrence

```
fix shuffle (A : Type) (xs ys : list A) {struct xs}
    : list A :=
  match xs with
  | nil => ys
  | x :: xs0 =>
      match ys with
      | nil => xs
      | y :: ys0 => x :: y :: shuffle A xs0 ys0
      end
  end
```

Une seule *variable de récurrence*

Vraie double récurrence

```
Fixpoint merge (xs ys:(list nat)) : (list nat) :=
  match xs, ys with
    nil, _ => ys
  | _, nil => xs
  | x::xs', y::ys' =>
    if (x <=? y) then x::(merge xs' ys)
    else y::(merge xs ys')
  end.

Error: Cannot guess decreasing argument of fix.
```

Deux variables de récurrence

Deux fonctions récursives

Deux points fixes

```
Definition merge :=
  (fix loop1 (xs ys:(list nat)) :=
     match xs with
       nil => ys
     | x::xs' =>
       (fix loop2 (ys:(list nat)) :=
          match ys with
            nil => xs
          | y::ys' =>
            if (x <=? y) then x::(loop1 xs' ys)
            else y::(loop2 ys')
          end)
        ys
     end).
```

Syntaxe

Point fixe local

```
Fixpoint merge (xs ys:(list nat)) : (list nat) :=
  let fix merge_aux ys :=
      match xs, ys with
      | nil, _ => ys
      | _, nil => xs
      | x::xs', y::ys' =>
        if (x <=? y) then x::(merge xs' ys)
        else y::(merge_aux ys')
      end
  in (merge_aux ys).
```

(Trop) stricte récurrence structurelle

Somme des éléments d'une liste avec accumulation «interne»

```
Fixpoint sum (ns:(list nat)) : nat :=
  match ns with
    nil => 0
  | n1::ns1 =>
    match ns1 with
      nil => n1
    | n2::ns2 => (sum ((n1+n2)::ns2))
    end
  end.
```

```
Recursive definition of sum is ill-formed.
[...]
Recursive call to sum has principal argument equal to
"n1 + n2 :: ns2" instead of one of the following variables:
"ns1" "ns2".
[...]
```

Et pourtant elle tourne

Et ne boucle pas

Borner (judicieusement) les appels récursifs

```
Definition sum (ns:(list nat)) : nat :=
  let fix loop (b:nat) (ns:(list nat)) :=
      match b with
        0 => 0
      | (S b') =>
        match ns with
          nil => 0
        | n::nil => n
        | n1::n2::ns' => (loop b' ((n1+n2)::ns'))
        end
      end
  in (loop (length ns) ns).
```

C'est la fonction attendue

```
Lemma sum_eq1 : (sum nil) = 0.
Proof. auto. Qed.

Lemma sum_eq2 : forall (n:nat), (sum (n::nil)) = n.
Proof. auto. Qed.

Lemma sum_eq3 : forall (n1 n2:nat) (ns:list nat),
  (sum (n1::n2::ns)) = (sum ((n1+n2)::ns)).
Proof. auto. Qed.
```

Mesure

Variant : (length ns)

```
Require Import Coq.Program.Wf.

Program Fixpoint sum (ns:(list nat))
        {measure (length ns)} :=
  match ns with
    nil => 0
  | n::nil => n
  | n1::n2::ns' => (sum ((n1+n2)::ns'))
  end.

sum has type-checked, generating 2 obligations
Solving obligations automatically...
sum_obligation_1 is defined
sum_obligation_2 is defined
No more obligations remaining
sum is defined
```

Récurrence structurelle en profondeur

On voudrait :

```
Fixpoint concat {A:Type} (xss:(list (list A))) : (list
  match xss with
    nil => nil
  | nil::xss' => (concat xss')
  | (x::xs')::xss' => x::(concat (xs'::xss'))
  end.
```

qui échoue

```
Error: Cannot guess decreasing argument of fix.
```

Prendre la bonne mesure

Dans

```
  | nil::xss' => (concat xss')
  | (x::xs')::xss' => x::(concat (xs'::xss'))
```

- soit la liste entière diminue
- soit la première liste diminue

```
Fixpoint deep_length {A:Type} (xss:(list (list A)))
    : nat :=
  match xss with
    nil => 0
  | xs::xss => (length xs)+(deep_length xss)
  end.
```

Appliquer la bonne mesure

```
Program Fixpoint concat {A:Type} (xss:(list (list A)))
        {measure (deep_length xss)}: (list A) :=
  match xss with
    nil => nil
  | nil::xss' => (concat xss')
  | (x::xs')::xss' => x::(concat (xs'::xss'))
  end.
```

Double récurrence

$$\begin{cases} \texttt{(ack 0 y)} & = & \texttt{(S y)} \\ \texttt{(ack (S x) 0)} & = & \texttt{(ack x (S 0))} \\ \texttt{(ack (S x) (S y))} & = & \texttt{(ack x (ack (S x) y))} \end{cases}$$

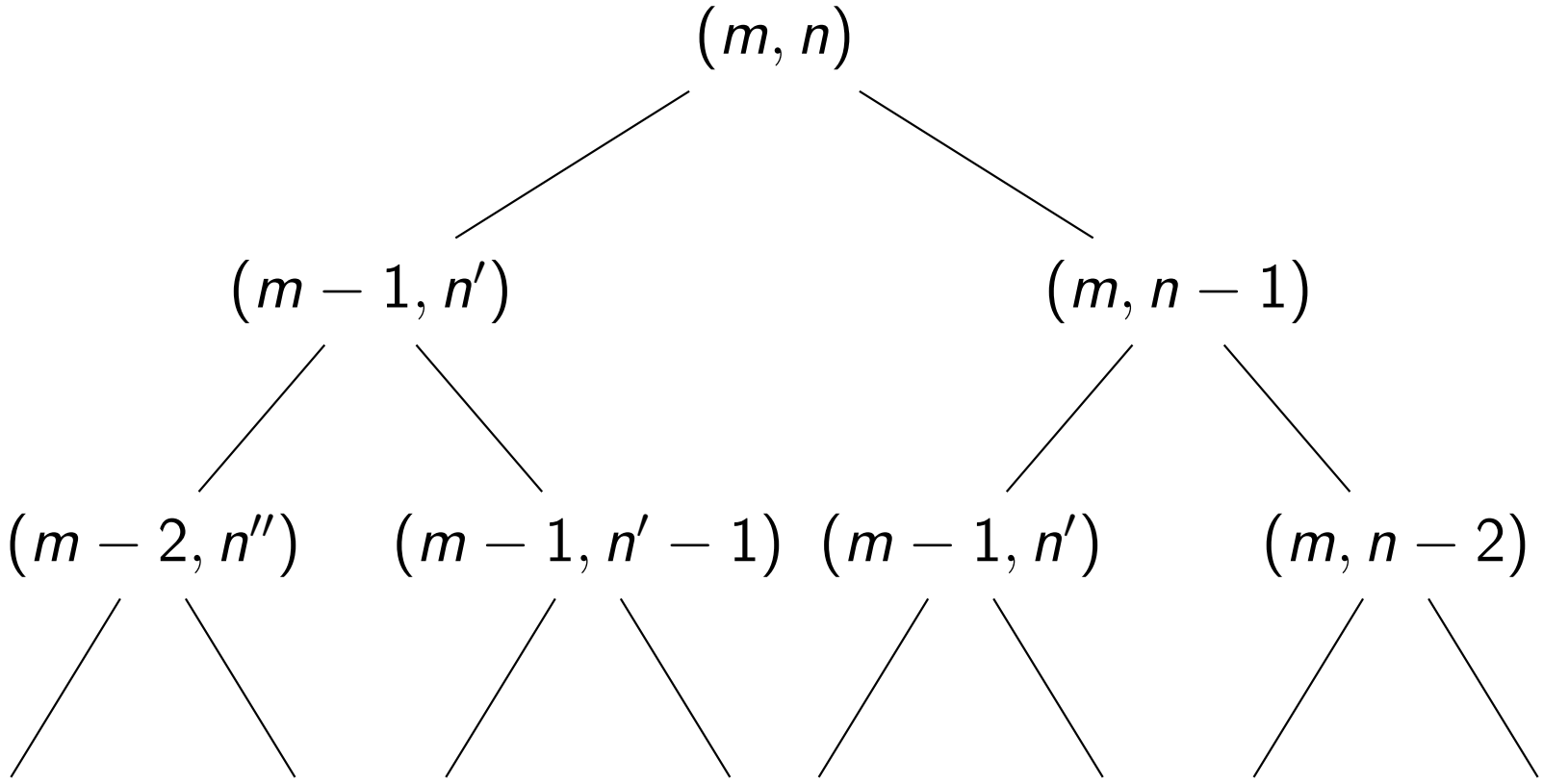
Terminaison :

- soit `x` décroît
- soit `y` décroît quand `x` reste constant

Ordre lexocigraphique sur `nat * nat`

Ordre lexicographique

Ordre bien fondé



Pas de branche infinie

Accessibilité

Bonne fondation en Coq (`Coq.Init.Wf`)

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

Definition well_founded := forall a:A, Acc a.
```

Intuitivement :

- `x` est *accessible* si tous ses prédécesseurs (`(R y x)`) le sont
- les éléments minimaux sont trivialement accessible (`not (R y 0)`)

Bonne fondation de l'ordre lexicographique

Ordre lexicographique sur `nat*nat`

```
Definition lt2 (xy1:(nat*nat)) (xy2:(nat*nat)) : Prop :
  match xy1,xy2 with
    (x1,y1),(x2,y2)
     => (x1 < x2) \/ ((x1 = x2) /\ (y1 < y2))
  end.

Lemma Acc_lt2 : forall (x y:nat), (Acc lt2 (x,y)).
```

Par double induction sur `x`, puis, dans chaque cas, `y`.

```
Theorem wf_lt2 : (well_founded lt2).
```

La définition

Décurryfication `ack : (nat * nat) -> nat`

```
Program Fixpoint ack (xy:nat*nat) wf lt2 xy : nat :=
  match xy with
    (0, y) => (S y)
  | (S x, 0) => ack (x, S 0)
  | (S x, S y) => ack (x, ack (S x, y))
  end.
```

Obligations de preuves

(Parfois obscures)

Réarrangment de structure

On voudrait faire :

```
Fixpoint list_of_btree A:Type (bt:btree A) : (list A) :
  match bt with
    Empty => nil
  | Node Empty x bt' => x::(list_of_btree bt')
  | Node (Node bt1 x1 bt2) x2 bt3
    => (list_of_btree (Node bt1 x1 (Node bt2 x2 bt3)))
  end.
```

Qu'est-ce qui décroît ?

- arbre entier dans le cas `(Node Empty x bt')`
- sous-arbre gauche dans le cas
  `(Node (Node bt1 x1 bt2) x2 bt3 )`

Ordre lexicographique
(taille de l'arbre, taille du sous arbre gauche)

Un ordre sur les arbres

```
Fixpoint size A:Type (bt:btree A) : nat :=
  match bt with
    Empty => 0
  | Node bt1 _ bt2 => S(size bt1 + size bt2)
  end.

Definition size2 A:Type (bt:btree A) : nat*nat :=
  match bt with
    Empty => (0,0)
  | Node bt1 _ _ => (size bt, size bt1)
  end.

Definition lt_btree A:Type (bt1 bt2:btree A) : Prop :=
  lt2 (size2 bt1) (size2 bt2).
```

Sa bonne fondation

```
Lemma Acc_Empty : forall (A:Type),
    (Acc lt_btree (Empty:btree A)).

Lemma Acc_Node : forall (A:Type) (x:A) (bt1 bt2:btree A
    (Acc lt_btree bt1) -> (Acc lt_btree bt2)
    -> (Acc lt_btree (Node bt1 x bt2)).

Lemma Acc_lt_btree : forall (A:Type) (bt:btree A),
    (Acc lt_btree bt).

Lemma wf_lt_btree A:Type :
    well_founded
    (fun (bt1 bt2:btree A) => lt_btree bt1 bt2).
```

Assez technique (en chantier :)

La fonction

```
Variable A:Type.
Program Fixpoint list_of_btree (bt:btree A)
        {wf lt_btree bt}: (list A) :=
  match bt with
    Empty => nil
  | Node Empty x bt' => x::(list_of_btree bt')
  | Node (Node bt1 x1 bt2) x2 bt3
    => (list_of_btree (Node bt1 x1 (Node bt2 x2 bt3)))
  end.
```

Obscur problème de typage...

Obligations

1. `(lt_btree bt' (Node Empty x bt'))`
2. `(lt_btree (Node bt1 x1 (Node bt2 x2 bt3))`
   `(Node (Node bt1 x1 bt2) x2 bt3))`
3. `(well_founded`
   `(MR lt_btree (fun arg : btree A => arg)))`