Master Parisien de Recherche en Informatique
cours 2-7-2 : assistants de preuve

Preuve de programmes fonctionnels (2)

Jean-Christophe Filliâtre

2005–2006

Récursion générale

pas toujours aisé de se ramener à **récursion structurelle** sur l'un des arguments ; plusieurs techniques présentées ici :

1. Récursion bornée
2. Fonctions récursives bien fondées
3. Récursion générale par itération
4. Récursion sur un prédicat ad-hoc

plus d'info : le **Coq'Art** de Bertot et Castéran

1. Récursion bornée

On ajoute un **argument artificiel** à la fonction
puis récurrence structurelle vis-à-vis de cet argument

généralement un entier qui **limite le nombre d'appels récursifs**

fonction définie par cas sur cet argument artificiel
le cas de base n'est pas nécessairement significatif
(car non nécessairement atteint)

Exemple : division euclidienne

par soustractions successives

```
let rec div m n =
  if n <= m then
    let (q,r) = div (m-n) n in (q+1,r)
  else
    (0,m)
```


Division avec un argument supplémentaire

```
Fixpoint div_aux (b m n : nat) {struct b} : nat*nat :=
  match b with
  | 0 =>
     (0,0)
  | S b' =>
     if le_gt_dec n m then
       let (q,r) := div_aux b' (m-n) n in
       (S q, r)
     else
       (0, m)
  end.
```


Correction

```
Theorem div_aux_correct : ∀b m n,
  m <= b → 0 < n →
  let (q,r) := div_aux b m n in
  m = q * n + r ∧ r < n.
```

par **récurrence** sur `b` (suit la définition de `div_aux`)

Instanciation de l'argument supplémentaire

définition d'une fonction `div` fortement spécifiée

```
Definition div : ∀m n, 0 < n→
  { q:nat & { r:nat | m=q*n+r ∧ r<n }}.
Proof.
  refine (fun m n (h:0<n) ⇒
    let p := div_aux m m n in
    existS (fun q ⇒ { r:nat | m=q*n+r ∧ r<n })
           (fst p) (exist _ (snd p) _)).
  ...
```


Avantages et inconvénients

Inconvénients

- calculs supplémentaires (non supprimés par l'extraction)
- la fonction obtenue est différente de la fonction souhaitée
- la borne doit être déterminée lors de l'appel

Avantage

- le calcul peut se faire facilement dans Coq (cf tactiques réflexives)

Calculs dans Coq

```
Time Eval lazy beta delta iota zeta in
  (div_aux 2000 2000 31).
     = (64, 16) : (nat * nat)%type
Finished transaction in 1. secs (0.23u,0.s)
```

div s'évalue également mais il faut prendre soin de **ne pas évaluer les termes de preuve**

```
Time Eval lazy beta delta iota zeta in
  match div 2000 31 (lt_O_Sn 30) with
   existS q (exist r _) ⇒ (q,r)
  end.
     = (64, 16) : (nat * nat)%type
Finished transaction in 0. secs (0.6u,0.s)
```

`Eval compute` prendrait **des heures !**

2. Récursion bien fondée

entrevue brièvement dans le cours précédent ; ici en détail

1. Relations bien fondées
2. Construction de relations bien fondées
3. Récursion bien fondée
4. Exemple : division euclidienne
5. Récursions imbriquées

2.1 Relations bien fondées

définies en utilisant la notion d'éléments **accessibles**

intuition

> *un élément x est accessible s'il n'appartient pas à une chaîne infinie décroissante (il n'y a pas de (u_n) telle que $u_0 = x$ et $R\ u_{i+1}\ u_i$ pour tout i)*

si une fonction récursive est telle que ses appels récursifs suivent une chaîne décroissante, alors elle termine lorsqu'elle est appliquée à un élément accessible

relation bien fondée = relation pour laquelle **tous** les éléments sont accessibles

Accessibilité

```
Inductive Acc (A : Set) (R : A → A → Prop) : A → Prop :=
  Acc_intro : ∀x : A, (∀y : A, R y x → Acc R y) → Acc R x.
```

autrement dit : une preuve de « x accessible » c'est une preuve que tous les prédécesseurs de x sont accessibles

où commence-t-on ? les éléments minimaux n'ont pas de prédécesseurs
⇒ on élimine l'absurde

Exemple

tous les entiers naturels sont accessibles pour `lt`

```
Theorem lt_Acc : ∀n:nat, Acc lt n.
```

par récurrence sur n

- $n = 0$:
  tout y tel que $y < 0$ est accessible ⇒ absurde
- n accessible ⇒ $S\ n$ accessible :
  soit y tel que $y < S\ n$; par cas
  - soit $y < n$ et alors y accessible par HR (car y préd. de n)
  - soit $y = n$ et alors y accessible par HR

Acc_inv

en pratique, on utilise souvent le résultat suivant

```
Lemma Acc_inv :
  ∀(A : Set) (R : A → A → Prop) (x : A),
    Acc R x → ∀y : A, R y x → Acc R y.
```

i.e. tout prédécesseur d'un élément accessible est accessible

Relation bien fondée

```
Definition well_founded (A : Set) (R : A → A → Prop) :=
  ∀a : A, Acc R a
```

exemple

```
Theorem lt_wf : well_founded lt.
exact lt_Acc.
```

abus de langage : c'est en fait la définition de **relation nœthérienne** (qui coïncide en logique classique avec celle de relation bien fondée)

Récursion structurelle comme cas particulier

de manière générale, la relation qui lie tout élément d'un type inductif à ses sous-termes stricts est bien fondée (preuve par récurrence structurelle)

2.2 Constructions de relations bien fondées

bibliothèque standard Coq : collection de résultats sur les relations bien fondées

exemple : la clôture transitive d'une relation BF est BF

```
wf_clos_trans:
  ∀(A : Set) (R : A → A → Prop),
  well_founded R → well_founded (clos_trans A R)
```

exemple : relation BF sur A à partir d'une relation BF sur B et $f : A \to B$

```
wf_inverse_image:
  ∀(A B : Set) (R : B → B → Prop) (f : A → B),
  well_founded R → well_founded (fun x y : A ⇒ R (f x) (f y))
```


2.3 Récursion bien fondée

on va effectuer une **récurrence structurelle** sur la preuve d'accessibilité

en effet

```
Acc_inv :
  ∀(A : Set) (R : A → A → Prop) (x : A),
    Acc R x → ∀y : A, R y x → Acc R y
:=
fun (A : Set) (R : A → A → Prop) (x : A) (H : Acc R x) ⇒
  match H in (Acc _ a) return (∀y : A, R y a → Acc R y) with
  | Acc_intro x0 H0 ⇒ H0
  end
```

la preuve $H0 : Acc\ R\ y$ est structurellement plus petite que $H : Acc\ R\ x$

Appplication : un opérateur de point fixe

on en déduit un opérateur de point fixe

```
Fixpoint Acc_iter
  (A : Set) (R : A → A → Prop) (P : A → Type)
  (F : ∀x : A, (∀y : A, R y x → P y) → P x)
  (x : A) (H : Acc R x) {struct H} : P x
:=
F x (fun (y : A) (h : R y x) ⇒ Acc_iter y (Acc_inv x H y h))
```

avec `Acc_inv x H y h` structurellement plus petit que `H`

well_founded_induction

```
Definition well_founded_induction
  (A : Set) (R : A → A → Prop) (Rwf : well_founded R)
  (P : A → Set)
  (F : ∀x : A, (∀y : A, R y x → P y) → P x)
  (a : A) : P a
:=
  Acc_iter P F (Rwf a)
```

car `Rwf a : Acc R a`

on a aussi `well_founded_ind` dans la sorte `Prop` (pour faire des **preuves** par récurrence bien fondée)

Contenu calculatoire

```
Extraction Acc_iter.

(** val acc_iter :
      ('a1 → ('a1 → __ → 'a2) → 'a2) → 'a1 → 'a2 **)

let rec acc_iter f x =
  f x (fun y _ → acc_iter f y)
```

c'est un **opérateur de point fixe**, similaire à

```
  let rec y f x = f (y f) x
```


2.4 Exemple : division euclidienne

$$\forall m\ \forall n,\ 0<n \to \underbrace{\{\ q : nat\ \&\ \underbrace{\{\ r : nat \mid \underbrace{m = q * n + r \land r < n}_{P''\ m\ n\ q\ r}\ \}}_{P'\ m\ n\ q}\ \}}_{P\ m}$$

```
Definition div : ∀m, P m
  := well_founded_induction_type lt_wf P div_F.
```


La fonctionnelle div_F

```
Definition div_F : ∀x, (∀y, y<x → P y) → P x.

unfold P.
refine (fun m div_rec n Hlt ⇒
  if le_gt_dec n m then
    let (q,h) := div_rec (m-n) _ n _ in
    let (r,_) := h in
    existS (P' m n) (S q)
      (exist (P'' m n (S q)) r _)
  else
    existS (P' m n) O
      (exist (P'' m n O) m _))
;
unfold P''; auto with arith.
```

un seul but

Difficulté

raisonner sur une fonction définie avec `well_founded_induction` est difficile

⇒ privilégier une spécification forte (comme ici)

2.5 Récursions imbriquées

$$f(x) = f(g(f(y)))$$

on utilise encore `well_founded_induction`

mais la spécification de f doit être suffisamment forte pour exprimer que l'argument de l'appel récursif (ici $g(f(y))$) est plus petit que l'argument initial (x)

Exemple

$$\begin{aligned} f(0) &= 0 \\ f(x+1) &= 1 + f\left(\lfloor \tfrac{x}{2} \rfloor + f\left(\lfloor \tfrac{x}{2} \rfloor\right)\right) \end{aligned}$$

on donne à f la spécification

```
∀x, { v:nat | v <= x }
```

on se donne

```
Parameter div2 : nat → nat.

Axiom double_div2_le :
  ∀x, div2 x + div2 x <= x.

Lemma f_lemma :
  ∀x v, v <= div2 x → div2 x + v <= x.
```


La fonctionnelle

```
Definition nested_F :
  ∀x,
    (∀y, y<x → { v:nat | v<=y } ) →
  { v:nat | v<=x }.

refine
  (fun x ⇒
     match x return (∀y, y<x → { v:nat | v<=y } ) →
                    { v:nat | v<=x } with
     | O ⇒
        (fun _ ⇒ exist _ O _)
     | S x' ⇒
        (fun f ⇒
           let (v,hv) := f (div2 x') _ in
           let (v1,hv1) := f (div2 x' + v) _ in
           exist _ (S v1) _) end);
```


Le point fixe

```
Definition nested_f :=
  well_founded_induction Wf_nat.lt_wf
    (fun x ⇒ { v:nat | v <= x }) nested_F.
```


3. Récursion par itération

lorsqu'il n'y a pas de récusion imbriquée

3 étapes :

1. déterminer la fonctionnelle associée à la fonction récursive
2. prouver que la fonction récursive termine (en utilisant seulement la fonctionnelle)
3. construire la fonction récursive

3.1 Fonctionnelle associée à une définition récursive

définition récursive

$$f(x) = e$$

f et x liés dans e $\Rightarrow$ fonctionnelle F telle que

$$f(x) = F(f, x)$$

Exemple : division euclidienne

```
div m n = if le_gt_dec n m then
            let (q,r) := div (m-n) n in (S q, r)
          else
            (0, m)
```

alors

```
Definition div_F
  (f : nat → nat → nat*nat) (m n : nat) :=
  if le_gt_dec n m then
    let (q,r) := f (m-n) n in (S q, r)
  else
    (0, m).
```


3.2 Terminaison

la fonctionnelle

$$F : (A \rightarrow B) \rightarrow (A \rightarrow B)$$

peut être **itérée**

si $g : A \rightarrow B$, alors $F(g) : A \rightarrow B$, $F(F(g)) : A \rightarrow B$, etc.
on note

$$F^k(g) = \underbrace{F(F(\ldots F(g)))}_{k \text{ fois}}$$

```
Fixpoint iter
  (A:Set) (n:nat) (F:A→A) (g:A) {struct n} : A :=
  match n with
  | O ⇒ g
  | S p ⇒ F (iter A p F g)
  end.
```

Suffisamment d'itérations...

si f est définie en a alors le calcul de $f(a)$ prend un certrain nombre d'itérations, soit p, et alors

$f(a) = F^k(g)(a)$ pour tout $k > p$ et toute fonction g (de bon type)

on va montrer le résultat

```
f_terminates :
  ∀x:A, { y:B | ∃p:nat,
                ∀(k:nat) (g:A→B),
                  p < k → iter k F g x = y }
```

par récurrence bien fondée (en utilisant une relation sur A)

Exemple

```
Lemma div_terminates :
  ∀m n, 0<n →
    { v:nat*nat |
      ∃p,
        ∀k, p < k →
          ∀g, iter k div_F g m n = v }.
```

by récurrence bien fondée sur m

```
induction m using (well_founded_induction lt_wf).
...
```


3.3 Construire la fonction f

f est alors obtenue en décomposant le résultat de f_terminates

```
Definition f (x:A) : B :=
  let (y,_) := f_terminates x in y.
```

exemple

```
Definition div m n (h:0<n) : nat * nat :=
  let (v,_) := div_terminates m n h in v.
```


3.4 Prouver l'équation de point fixe

on montre

```
Lemma f_fix_eq : ∀(x:A), f x = F f x
```

exemple

```
Lemma div_fix_eq : ∀m n (h:0<n),
  div m n h = if le_gt_dec n m then
                let (q,r) := div (m-n) n h in (S q, r)
              else
                (0,m).

  ∃p, ∀k, p<k → div m n h = iter k div_F g m n h
  ∃p', ∀k, p'<k → div (m-n) n h = iter k div_F g (m-n) n h
```

il suffit de prendre $k = S(S(max(p, p')))$

3.5 Utiliser l'équation de point fixe

...pour prouver des propriétés de la fonction f

par récurrence bien fondée sur l'argument

exemple

```
Theorem div_correct :
  ∀m n (h:0<n),
    let (q,r) := div m n h in
    m = q * n + r ∧ r < n.
Proof.
  induction m using (well_founded_ind lt_wf).
  intros; rewrite div_fix_eq.
  ...
```


Différence avec la récursion bornée

à l'extraction, le nombre d'itérations **disparaît**
car dans f_terminates on a mis $\exists p$ dans Prop

```
Extraction Inline div_terminates.
Extraction div.
(** val div : nat → nat → (nat, nat) prod **)

let rec div x n =
  match le_gt_dec n x with
    | Left → let Pair (q, r) = div (minus x n) n in
             Pair ((S q), r)
    | Right → Pair (O, x)
```


4. Récurrence sur un prédicat ad-hoc

récurrence bien fondée = récurrence sur la preuve d'un prédicat inductif, Acc

de manière générale, il est possible de définir des fonctions par **récurrence structurelle sur n'importe quel prédicat**

cependant la sorte Prop restreint les filtrages possibles ⇒ impossible de filtrer directement la preuve du prédicat ⇒ **théorème d'inversion**

4.1 Définition d'un prédicat ad-hoc

exemple

```
let rec log = function
  | S O → O
  | S (S p) → S (log (S (div2 p)))
```

prédicat ad-hoc

```
Inductive log_domain : nat → Prop :=
  | log_dom_1 :
      log_domain 1
  | log_dom_2 :
      ∀p, log_domain (S (div2 p)) →
          log_domain (S (S p)).
```

(note : on peut montrer $\forall n$, log_domain n)

4.2 Théorèmes d'inversion

```
Theorem log_domain_not_0 :
  ∀x, log_domain x → x <> 0.

Theorem log_domain_inv :
  ∀x p, log_domain x →
    x = S (S p) →
    log_domain (S (div2 p)).
Proof.
  ...
Defined.
```

prendre soin que la preuve obtenue soir **structurellement plus petite** que celle en argument

4.3 Définition de la fonction

```
Fixpoint log (x:nat) (h:log_domain x) {struct h}:nat :=
  match x as y return x=y → nat with
  | 0 ⇒ (fun h' ⇒ False_rec nat (log_domain_not_0 x h h'))
  | S 0 ⇒ (fun _ ⇒ 0)
  | S (S p) ⇒
     (fun h' ⇒ S (log (S (div2 p)) (log_domain_inv x p h h'))
  end (refl_equal x).
```

`log_domain_inv x p h h'` **plus petit que** `h`

4.4 Établir des propriétés de la fonction

par récurrence sur le prédicat

attention
il faut utiliser le principe de récurrence **maximal** (schéma dépendant)

n'est pas produit par défaut ; s'obtient avec la commande `Scheme`

```
Scheme log_domain_ind2 := Induction for log_domain Sort Prop.
```


Principe maximal (schéma dépendant)

```
Scheme log_domain_ind2 := Induction for log_domain Sort Prop.
```

```
log_domain_ind2 :
  ∀P : ∀n : nat, log_domain n → Prop,
     P 1 log_dom_1 →
     (∀(p : nat) (l : log_domain (S (div2 p))),
        P (S (div2 p)) l → P (S (S p)) (log_dom_2 p l)) →
     ∀(n : nat) (l : log_domain n), P n l
```


Utilisation du schéma maximal

montrons par exemple

$$2^{\log(x)} \leq x$$

```
Fixpoint two_power (n:nat) : nat := match n with
  | 0 ⇒ 1
  | S p ⇒ 2 * two_power p
  end.

Theorem pow_log_le :
  ∀x (h: log_domain x), two_power (log x h) <= x.
Proof.
  induction h using log_domain_ind2.
  ...
```


Remarque

bien entendu, on aurait pû donner à la fonction `log` une spécification forte