

# *Compiler des Polynômes en FoCaLize?*

Renaud Rioboo



Journée Ssurf 29 juin 2010

## *Pourquoi des polynômes ?*

- Historique : FoC, FoCaL, Zenon, FoCaLize ! 1997-2010
- Mathématiques et programmation raisonnées.
- Exemple pour dégager des traits de programmation.
- Modèle Ocaml, FoC : Calculemus 2000  
`poly_collections_oo.ml` daté du 24/05/2000
- Modèle FoCaL, Zenon :  
`polys_concrete.foc` daté du 10/06/2008
- Polynômes récursifs pas dans FoCaLize 1.0!  
`recursive_polys.fcl`

## Polynômes distribués

$\mathbf{A}$  un anneau,  $D$  des « degrés » ( $\mathbb{N}, \mathbb{N} \times \mathbb{N}, \dots$ )

$$\left\{ \begin{array}{l} \text{polynômes constants :} \\ \mathcal{P}_{0_D} = 0_{\mathcal{P}} \uplus \{(a, 0_D) \mid a \neq 0_{\mathbf{A}}\} \\ \\ \text{polynômes de degré } d : \\ \mathcal{P}_d = \{((d, a), p) \mid d \in D, a \in \mathbf{A}, p \in \mathcal{P}_{d'}, d' < d, a \neq 0_{\mathbf{A}}\} \end{array} \right.$$

$$\mathcal{P} = \uplus_{d \geq 0_D} \mathcal{P}_d \text{ bien défini}$$

```
type distributed_representation ('a, 'd) =  
  | Null  
  | NonNull ('a, 'd, distributed_representation ('a, 'd))  
;;
```

## Ocaml, FoC

- Un objet

« porte » les mathématiques de la donnée, c'est la **collection** :

```
class ['r,'d,'a,'b] algebre_indexee((r,d) : ('r*'d))=  
object(the_pols)
```

```
constraint 'r = ('a)#anneau_commutatif  
constraint 'd = ('b)#ordre_monomial
```

```
inherit ['r,'d,'a,'b,('a*'b) list] algebre_formelle_indexee(
```

- Le type des données manipulées est le dernier paramètre de la classe, c'est la **représentation**.
- La classe décrit l'objet, c'est l'**espèce**.

## *FoCaL, FoCaLize*

Les valeurs :

```
species Indexed_set (I is Ordered_set_with_zero,  
                    E is Setoid_with_zero) =  
  inherit Setoid_with_zero ;
```

```
representation = distributed_representation (E, I) ;  
let zero = Null ;
```

```
let monomial (x : E, d : I) : Self =  
  if E!is_zero (x)  
  then zero  
  else NonNull (x, d, Null) ;
```

## *Aspects ensemblistes*

Égalité : répartition des égalités sur les coefficients et les degrés :

proof of equal\_reflexive =

<1>1 prove equal(Null, Null)

by type distributed\_representation

definition of equal

<1>2 prove all x : Self, all cx : E, all dx : I,

equal(x, x) -> equal(NonNull(cx, dx, x),

NonNull(cx, dx, x))

by definition of equal

type distributed\_representation

property E!equal\_reflexive, I!equal\_reflexive

<1>3 prove all x : distributed\_representation(E, I),

!equal(x, x)

by step <1>1, <1>2 type distributed\_representation

<1>f conclude ;

Zenon fait l'induction structurelle!

## *Abstraire, curryfier les propriétés à démontrer!*

```
logical let equal_symmetry_prop(x : Self) : prop =  
  all y : Self, equal(x, y) -> equal(y, x);
```

```
proof of equal_symmetric =
```

```
  <1>1 prove equal_symmetry_prop(Null)
```

```
    ...
```

```
  <1>2 prove all x : Self, all cx : E, all dx : I,  
    equal_symmetry_prop(x) ->  
      equal_symmetry_prop(NonNull(cx, dx, x))
```

```
    ...
```

```
  <1>3 prove all x : distributed_representation(E, I),  
    equal_symmetry_prop(x)
```

```
    by step<1>1, <1>2 type distributed_representation
```

```
  <1>f qed by step <1>3
```

```
    definition of equal_symmetry_prop
```

```
;
```

## *Mettre un invariant ?*

Essayer d'établir un invariant prouvable ?

```
logical let correct_rep(p : distributed_representation(E, I)
  match p with
  | Null -> True
  | NonNull(c, d, Null) ->
    (I !geq(d, I!zero) && E!different(c, E!zero))
  | NonNull(c, d, pp) ->
    (I!gt(d, !degree(pp)) && E!different(c, E!zero))
;
```

## De l'arithmétique ?

```
let rec plus (s_1, s_2) = match s_1 with
| Null -> s_2
| NonNull (m1, d1, ss_1) -> match s_2 with
  | Null -> s_1
  | NonNull (m2, d2, ss_2) ->
    if I!lt (d1, d2)
    then (* d1 < d2 *)
      NonNull (m2, d2, !plus (s_1, ss_2))
    else
      if I!lt (d2, d1)
      then (* d2 < d1 *)
        NonNull (m1, d1, !plus (ss_1, s_2))
      else (* d2 = d1 total order *)
        let m = M!plus (m1, m2) in
        if M!is_zero (m)
        then !plus (ss_1, ss_2)
        else NonNull (m, d1, !plus (ss_1, ss_2));
```

## *Montrer l'arithmétique ?*

```
proof of zero_is_neutral =  
  <1>1 prove all x : Self, !equal(!plus(!zero, x), x)  
    assumed  
    (* by definition of zero, plus *)  
    (* property equal_reflexive *)  
    (* type distributed_representation *)  
  <1>2 prove all x : Self, !equal(!plus(x, !zero), x)  
    by step <1>1  
      property !plus_commutates, equal_transitive  
  <1>f conclude ;  
  
proof of opposite_is_opposite = assumed ;
```

## Polynômes récurrents

$\mathbf{A}$  un anneau,  $\{v_n\}_{n \in \mathbb{N}_{\geq 1}}$ , des “variables”

$$\left\{ \begin{array}{l} \text{polynômes constants :} \\ \mathcal{P}_0 = \mathbf{A} \\ \\ \text{polynômes en } v_i : \\ \mathcal{P}_{i,0_D} = 0_{\mathcal{P}_i} \uplus \{(c, 0_D), c \in \mathcal{P}_j, j < i, c \neq 0_{\mathbf{A}}\} \\ \mathcal{P}_{i,d} = \{(c, d), p\} c \in \mathcal{P}_j, j < i, c \neq 0_{\mathbf{A}}, d > 0_D, p \in \mathcal{P}_{i,d'}, d' < d \\ \\ \text{polynômes non constants en } v_i : \\ \mathcal{P}_i = (v_i, \uplus_{d > 0_D} \mathcal{P}_{i,d}) \end{array} \right.$$

$$\mathcal{P} = \uplus_{i \geq 0_{\mathbb{N}}} \mathcal{P}_i \text{ bien défini}$$

## Représentation Récursive

Un polynôme a une variable principale ( $x$ ) et ses coefficients sont récursivement des polynômes sans cette variable :

L'anneau : 'a, les degrés : 'd

```
type recursive_representation('a, 'd) =  
  | Base ('a)  
  | Composed  
    (string,  
     distributed_representation  
      (recursive_representation('a, 'd), 'd))  
;;  
let one = Base(1) ;;  
let y_2 = Composed("y", NonNull(one, 2, Null)) ;;  
let x_2_plus_y_2 =  
  Composed("x", NonNull(one, 2, NonNull(y_2, 0, Null)))
```

Transparence de la représentation !

## *Porter les opérations ? Ocaml*

```
class ['r,'a,'v,'b,'c,'d] algebre_indexee_recursive
  ((rng,v,f) : ('r*'v*('c->'d)))=
object(rec_rng : 'rng)
  constraint 'c = (('a,'b)rec_struct)#anneau_commutatif
  constraint 'd = ('c,
                    'v,
                    (('a,'b)rec_struct as 'rs),
                    'b,
                    (('rs*'b)list)
                    )#algebre_formelle_indexee
inherit ['r,'a,'v,'b,'d]algebre_recursive(rng)

val mutable up_dom : ('d)option = None
method up_dom =
  match up_dom with
  | None -> let t = f rec_rng in let () = up_dom <- Some
  | Some t -> t
```

## *On s'appuie sur les méthodes de `up_dom`*

On appelle la méthode `output` de la collection `rec_rng#up_dom`

```
method print =
```

```
  let up_out = (rec_rng#up_dom)#output
```

```
  and under_print x = rng#print x
```

```
  in
```

```
  function
```

```
    | Base a -> under_print a
```

```
    | Composed (v,p) -> up_out(p,v)
```

Le type de la

méthode `up_dom` de `rec_rng (Self)` est bien 'd (les polynômes récur­sifs)!

```

val multiplie = fun me_multiplie mod_mult mult ->
  function
  | ((Base a1), (Base a2)) -> Base (under_multiplie(a1,a2))
  | ((Base a1), aa2) -> me_multiplie(a1,aa2)
  | (aa1 , (Base a2)) -> me_multiplie(a2,aa1)
  | (((Composed (v1,a1)) as aa1), ((Composed (v2,a2)) as aa2)) ->
    if (v1 < v2)
    then Composed (v2,mod_mult(aa1,a2))
    else
      if (v2 < v1)
      then Composed (v1,mod_mult(aa2,a1))
      else Composed (v1,mult(a1,a2))

```

```

method multiplie =
  multiplie
  rec_rng#module_multiplie
  (rec_rng#up_dom)#module_multiplie
  (rec_rng#up_dom)#multiplie

```

## *FoCaL compilation objet : Ocaml 3.09*

```
let rec
  updom is formal_polynomials_commutative_ring(self,v)=
    distributed_polynomials_com_ring(self,v)
  and print(p in self) =
    match p with
    | #Base(n) -> r!print(n)
    | #Composed(s,p) -> #sc(#sc("(",self!updom!output(p,s)),")",)
    end
  and plus(p,q) = ...
  and mult(p,q) = ...
```

La méthode `output` de `self !updom` rappelle `self !print!`

Le code de `plus` contient `self !updom !plus`, celui de `mult` contient `self !updom !module_mult` et `self !updom !mult`.

## *FoCaLize ?*

- Casser la récursion liée à `updom`.
- Abstraire les méthodes de `updom` au top level.
- `i_print` : impression des degrés.
- `c_print` : impression des coefficients.

```
let distr_output(i_print, c_print) =  
  let rec my_out(p, c, d, v) = match p with  
  | Null -> c_print(c) ^ v ^ "**" ^ i_print(d)  
  | NonNull(cc, dd, pp) ->  
    c_print(c) ^ v ^ "**" ^ i_print(d) ^ "+" ^  
      my_out(pp, cc, dd, v)  
  in function p -> function v -> match p with  
  | Null -> "0"  
  | NonNull(cp, dp, rp) -> my_out(rp, cp, dp, v)  
; ;
```

## *Encapsuler dans une espece ?*

Récursion entre l'impression des coefficients (`my_print`) et l'impression des polynômes distribués (`d_out`).

```
species Recursive_indexed_set (E is Setoid_with_zero,  
                               I is Ordered_set_with_zero)
```

```
inherit Setoid_with_zero ;
```

```
representation = recursive_representation (E, I) ;
```

```
let print =
```

```
  let
```

```
    rec my_print(x) = match x with
```

```
      | Base(xx) -> E!print(xx)
```

```
      | Composed(v, p) -> d_out(p, v)
```

```
    and d_out(p, v) = distr_output(I!print, my_print, p, v
```

```
    in function x -> my_print(x) ;
```





