Factorisation de polynômes sur les corps finis

Louis Mandel

Rapport de stage du Magistère d'Informatique Appliquée de l'Ile de France

Encadrant : Renaud Rioboo
Avril-Septembre 2001

Laboratoire d'Informatique de Paris 6

Table des matières

Introduction

Ce document retrace le travail que j'ai effectué au cours de mon stage au LIP6 d'Avril à Septembre 2001.

Ce stage s'inscrit dans le cadre de la deuxième année du Magistère d'Informatique Appliquée de l'Ile-de-France (MIAIF). Le magistère est une formation en trois ans, qui permet à une trentaine d'étudiants de suivre un enseignement en informatique très large.

Le projet FOC (Formel + OCaml + Coq) est un projet du LIP6, plus précisément du pôle MFI (Méthodes Formelles et Implantations) composé des thèmes SPI (Sémantique Preuve et Implantation) et CalFor (Calcul Formel). FOC a pour but la réalisation d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel.

Le but de mon stage est d'implanter un algorithme de factorisation des polynômes au sein de l'atelier de Calcul Formel Certifié FOC.

Chapitre 1

Le projet FOC et mon stage

1.1 Le laboratoire d'informatique de l'université Paris 6 (LIP6)

1.1.1 Le laboratoire d'informatique

Le LIP6 est situé dans un immeuble de Commissariat à l'Énergie Atomique dans le 15ème arrondissement près du Champ de Mars à Paris. Le laboratoire d'informatique de l'université Paris VI Pierre et Marie Curie regroupe 9 équipes d'enseignants-chercheurs par thèmes. Ces thèmes sont :

- Algorithmique numérique et parallélisme (ANP), resp. R. Alt & M. Minou
- Apprentissage et acquisition de connaissances (APA), resp. B. Bouchon-Meunier
- Architecture, systèmes intégrés et micro-électronique (ASIM), resp. A. Greiner
- Calcul formel (CalFor), resp. D. Lazard
- Réseaux et performances (RP), resp. S. Fdida
- Objets et agents pour les systèmes d'information et de simulation (OASIS), resp. J.-P. Briot
- Sémantique, preuve et implantation (SPI), resp. Th. Hardin
- Systèmes répartis et coopératifs (SRC), resp. C. Girault
- Systèmes d'aide à la décision et à la formation (SysDeF), resp. J.-Y. Jaffray

1.1.2 Le pôle MFI

Le pôle MFI regroupe deux thèmes qui ont entre autres en commun l'initiative du projet FOC. Mais pour mieux comprendre leur travail commun décrivons brièvement leurs activités propres.

- Le thème SPI :
  Le thème Sémantique, Preuve et Implantation est une équipe de chercheurs en programmation. Le nom même du thème reflète les trois composantes essentielles des activités de recherche :

  1. La recherche en sémantique pour fixer le sens des formalismes et des langages employés.
  2. Le développement de preuves certifiant la pertinence des composants réalisés.
  3. L'implantation aussi bien de langages de spécification que de langages de programmation et d'outils d'aide à la preuve.

- Le thème CalFor :
  L'activité du thème est centrée autour du Calcul Formel, et plus spécialement du problème fondamental de la résolution des systèmes d'équations polynomiales. Il existe des logiciels efficaces actuellement pour résoudre des systèmes ayant un nombre fini de solutions. Dans le cas plus difficile où l'ensemble des solutions est infini, les nouveaux algorithmes découverts

et implantés par l'équipe CalFor constituent une étape importante vers une résolution satisfaisante. Il est à noter que ces derniers travaux ont un prolongement en algèbre différentielle, ce qui a conduit l'équipe Calfor à collaborer avec le LIFL (Lille) et le LMC (Grenoble). Ces logiciels ont des applications potentielles dans les domaines scientifiques les plus variés. Ils ont été utilisés avec succès par les membres d'une équipe en cinématique des robots et en compression d'images (brevet déposé).

1.2 Le projet FOC

Le projet FOC (Formel + OCaml + Coq) dont le responsable est Thérèse Hardin a pour but la réalisation d'un environnement de calcul formel certifié. Il s'appuie sur OCaml pour la partie calculatoire et Coq pour la partie certification. FOC est proche d'Axiom dans la modélisation des structures mathématiques, mais en diffère par le fait que FOC est certifié.

1.2.1 Les membres

La composistion de l'équipe du projet FOC est la suivante :

- Pour le LIP6
  - de l'équipe CALFOR
    - Renaud Rioboo
    - Valérie Ménissier-Morain
  - de l'équipe SPI
    - Thérèse Hardin
    - Sylvain Boulmé
    - Virgile Prévosto
- De l'équipe Moscova de INRIA
  - Damien Doligez
- De l'équipe CPR du CEDRIC (CNAM)
  - Véronique Viguié Donzeau-Gouge
  - Catherine Dubois.

Le projet fait également partie de l'action coopérative CFC (Calcul Formel Certifié) de l'INRIA avec les équipes Coq, Cristal, Croap et Lemme de l'INRIA.

1.2.2 Le langage FOC

Le langage FOC permet à l'utilisateur d'écrire un source contenant à la fois les énoncés des propriétés et les algorithmes. Un programme OCaml est généré à partir de ce fichier source. L'utilisateur peut construire interactivement la preuve de ses énoncés, preuve qui est vérifiée à l'aide de Coq.

Espèces et collections

En FOC, nous manipulons deux types d'objets : les espèces et les collections.

Une espèce décrit une structure mathématique ; il y a des espèces pour définir ce qu'est un groupe, un anneau ou un corps. Une espèce a les composantes suivantes :

- un support,
- des propriétés et théorèmes,
- et des opérations.

Le support définit la représentation des éléments de l'espèce. Le support est un type qui peut être polymorphe. Les propriétés et théorèmes peuvent être seulement énoncés ou démontrés. De même les opérations peuvent être déclarées ou implémentées. La création d'espèce peut se faire par extension, raffinement, redéfinition ou instanciation du support à travers une relation d'héritage multiple. Les espèces peuvent être paramétrées par des collections ou des entités (élément d'une collection).

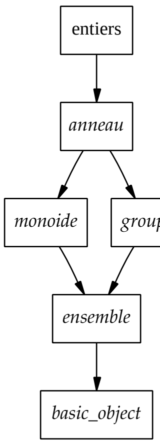
Les collections représentent les ensembles mathématiques. Toutes les méthodes sont implantées et les théorèmes sont démontrés dans une collection. Les collections ne peuvent plus être étendues ou modifiées.

Un exemple

En FOC toutes les espèces sont issues de l'espèce `basic_object`

```
species basic_object =
 rep T;
;;
```

qui défini juste un type support, qui est un type abstrait. Nous allons dans cette exemple construire un anneau en partant de `basic_object` et en suivant la structure suivante.



On définit un ensemble :

```
species ensemble inherits basic_object =
  sig egal in  self->self->bool;
  sig element in self;
  let different (x,y) = not_b(self.egal(x,y));
;;
```

qui a deux déclarations d'opération et une implantation d'opération. Si on ajoute une loi multiplicative et un élément neutre pour cette loi alors on obtient un monoïde.

```
species monoide inherits ensemble =
  sig multiplie in self -> self -> self;
  sig un in self;
;;
```

On peut de même définir un groupe.

```
species groupe inherits ensemble =
  sig zero in self;
  sig plus in self -> self -> self;
  sig oppose in self -> self;
```

```
  let moins(x,y) = self.plus(x,self.oppose(y));
;;
```

Grâce à l'héritage multiple, on peut créer l'espèce anneau :

```
species anneau inherits monoide, groupe =
;;
```

Maintenant, pour pouvoir faire des calculs sur l'anneau des entiers, on crée une collection qui l'implante.

```
collection entiers implements anneau =
  rep T = int;
  let zero = 0;
  let un = 1;
  let plus(m,n) = int_plus(m,n);
  let oppose(n) = int_opp(n);
  let multiplie(m,n) = int_mult(m,n);
  let egal(x,y) = struct_eq(x,y);
  let of_int(x in int) in self = x;
;;
```

Les fonctions `int_plus`, `int_mult` et `struct_eq` sont directement importées de OCaml par l'intermédiaire de fichiers `.fml`.

On peut constater que l'on a pu définir simplement et en restant près des concepts, des structures algébriques de base.

Les listes En FOC, on a un type `list('a)`, qui est la liste des éléments de type `'a`. `Nil` est la liste vide et `Cons(hd,tl)` est le constructeur de liste.

Les couples Pour écrire un couple en FOC, on doit utiliser la fonction `crp`. Donc le couple (a, b) s'écrira `crp(a,b)` Il y a les fonctions d'accès `first` et `scnd` pour récupérer respectivement la valeur du premier ou du second élément du couple.

Pour plus de précisions, la grammaire du langage est présentée dans [**?**].

1.2.3 La librairie OCaml

La compilation de FOC vers OCaml se base sur l'expérience acquise lors de l'écriture de la librairie en OCaml. Il y a eu cinq prototypes qui ont été développés pour pouvoir définir le cahier des charges de la programmation de la bibliothèque. Le développement en OCaml a par exemple permis de choisir une modélisation par objets et modules, ce choix a été expliqué dans [**?**]. Actuellement, la librairie est developpée parallèlement en FOC et en OCaml.

1.2.4 La structure de FOC

L'organisation entre les espèces de FOC reprend les concepts de l'algèbre. La librairie est composée de plus de 80 espèces. L'organisation entre les espèces de base est représentée dans le graphe de l'annexe A.

Ce graphe est issu d'une analyse du fichier source OCaml générant du code DOT. Le logiciel `dot` s'occupe de tracer le graphe qu'on lui décrit dans le fichier source. On peut trouver `dot` à l'adresse suivante : `http://www.research.att.com/sw/tools/graphviz/`.

L'outil d'analyse de code prend en entrée des fichiers sources OCaml et donne en sortie un fichier `dot` qui représente le graphe d'héritage de classe. On peut choisir deux types de formalisme.

- Soit comme ce qui est présent dans rapport, on donne seulement le nom des classes (en italique pour les classes abstraites), et le nom est dans un ovale si l'héritage de cette classe n'est pas défini.
- Soit une sortie UML.

On peut choisir de générer le graphe en postscript ou d'avoir une sortie HTML. Si on choisit la sortie HTML, il y a la possibilité de voir le code source de la classe en cliquant sur le noeud correspondant. J'ai développé cet outil durant ce stage et Virgile Prévosto en a fait un équivalent pour le code FOC.

1.3 Le sujet du stage

Le sujet du stage porte sur la factorisation des polynômes à coefficients sur un corps fini et les extensions algébriques dans le cadre du projet FOC. La factorisation est une fonctionnalité importante pour tout système de calcul formel. Et toutes les factorisations reposent sur la factorisation des polynômes à coefficients dans un corps fini. Les extensions algébriques permettent de tester la factorisation.

1.3.1 Travailler dans le projet FOC

La programmation dans le cadre du projet demande une grande dicipline. Il faut suivre le cahier des charges et la spécification formelle qui ont été réalisés. Les choix de conception doivent être respectés. Le code que j'ai écrit a du être testé et des mesures de performances réalisées. Une revue de code va être effectuée.

Ces étapes sont indispensables car finalement les preuves vont être réalisées.

Chapitre 2

Les entiers

Que l'on parle de degré de polynôme univarié, de caractéristique d'un corps, ces entités sont des entiers qui ont un certain nombre de propriétés. Nous voulons pouvoir utiliser ces entités sans avoir à choisir une représentation machine. Ceci est indipensable, car on veut pouvoir aussi bien définir un corps de caractéristique 5, qui se code très bien sur les `int` de OCaml, qu'un corps de caractéristique 10000000033, qui nécessite l'utilisation des entiers de `big_int` ou `gmp`. Le fait de vouloir donner des propriétés à ces nombres, nous oblige à encapsuler les entiers dans des espèces.

2.1 Les espèces

2.1.1 Coercion

L'espèce `entiers_base`

```
species entiers_base inherits monoide_additif_unitaire =
  sig to_int in self -> int;
;;
```

est à la base de toutes les espèces d'entiers. Elle hérite de l'espèce `monoide_additif_unitaire` pour avoir la méthode `of_int`. Avec cette méthode `of_int` et la méthode `to_int` qu'elle déclare, l'espèce `entiers_base` montre que toutes les collections d'entiers auront un fort lien entre elles à travers cette coercion explicite entre le type support et les `int` de OCaml.

2.1.2 Les naturels

L'espèce `naturels_ordonnes`

```
species naturels_ordonnes inherits ordre_monomial, entiers_base =
  sig successeur in self -> self;
  sig predecesseur in self -> partiel(self);
  sig random in self -> self;
;;
```

On ajoute un ordre sur les naturels et un élément minimal. Ces propriétés sont données par l'héritage de l'espèce `ordre_monomial`.

L'espèce `semi_anneau_naturels`

```
species semi_anneau_naturels inherits naturels_ordonnes, semi_anneau_euclidien =
;;
```

On donne en plus les propriétés de semi-anneau euclidien ; ce qui permet de calculer la division, le pgcd et beaucoup d'autres choses. Avec cette espèce, on a toutes les propriétés sur les entiers naturels.

2.1.3 Les entiers

L'espèce `entiers`

```
species entiers
  inherits groupe_additif_ordonne, anneau_euclidien, entiers_base =

  sig successeur in self -> self;
  sig predecesseur in self -> self;
;;
```

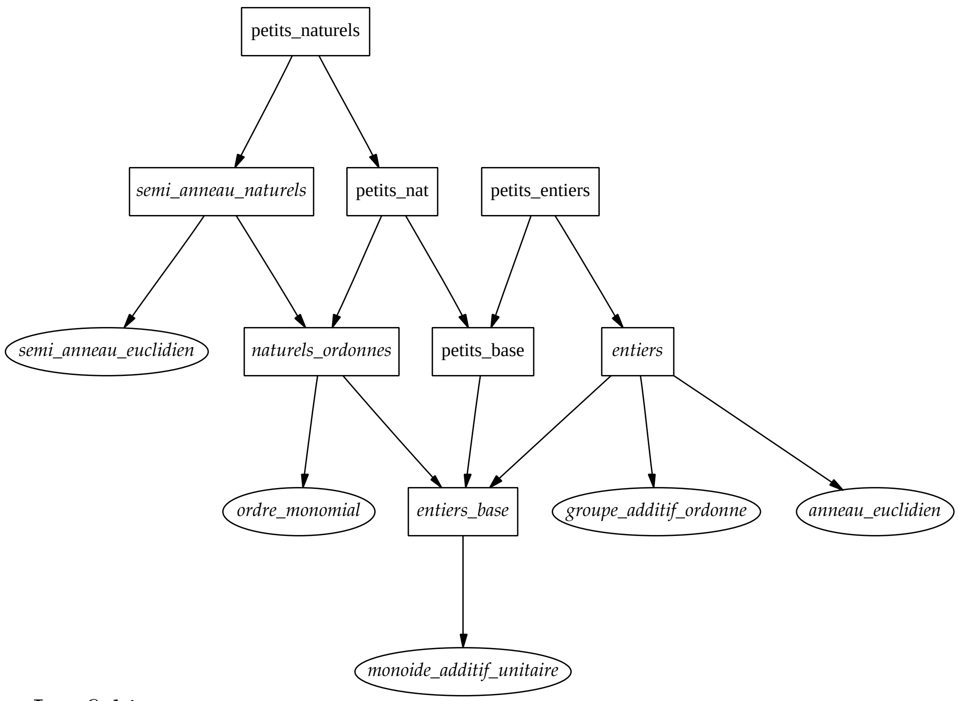
Les entiers ne sont pas construits à partir des naturels, car on ne veut pas hériter du minimum. Et on voit bien que la fonctiom `predecesseur` n'est pas du même type que dans les naturels car ici, il existe toujours un prédécesseur.

2.2 Les collections

A partir des espèces définies précédemment, il est très simple de définir les entiers implémantés par les `int` ou les `big_int`. Nous allons donc fixer le type support et implanter toutes les méthodes, mais nous n'allons pas faire de collections, pour permettre l'héritage.

2.2.1 L'architecture

On montre ici, comment on place l'implémentation des entiers avec des `int` par rapport aux espèces. Pour chaque implémentation, il faut reprendre la même structure.



Les fichiers `species_entiers.foc` et `collections_entiers_int.foc`

2.2.2 L'implantation

On prend pour exemple l'implantation avec les `int`.

```
species petits_base inherits entiers_base =
  rep T = int;

  let zero = 0;
  let un = 1;
  let is_zero(x in self) = struct_eq(x,0);
  let plus(m,n) =
    let res = int_plus(m,n) in
      if int_lt(res,0) then
       foc_error("Overflow")
      else res;
  let equal(x,y) = struct_eq(x,y);
  let is_un(x) = struct_eq(x,1);
  let to_int(x in self) in int = x;
;;
```

On fixe le type support au type `int`. Pour les méthodes, on utilise des fonctions importées de OCaml.

La méthode `to_int`, dans ce cas, change seulement le type de l'élément. Elle tansforme `x` qui était un élément de l'ensemble des `petits_base` sur les `int` en un simple `int`.

2.3 Les corps finis

Nous allons voir ici l'utilisation des entiers pour la définition des corps finis.

Nous allons noter $\mathbb{F}_q$ les corps finis a q éléments. Le cardinal q du corps est égal à p^n, où p est la caractéristique et n la dimension du corps. Nous allons donc définir les corps finis de la manière suivante en FOC.

```
species corps_fini (nat is semi_anneau_naturels)
  inherits corps =

  sig cardinal in nat;
  sig caracteristique in nat;
  sig dimension in nat;
  sig random in unit -> self;
;;
```

Donc un corps fini en FOC est un corps sur lequel on a des informations supplémentaires sur le nombre d'éléments. La représentation de ces informations est paramètrée par `nat` qui est un ensemble sur les naturels.

2.4 Les polynômes

Une autre utilisation des naturels est la représentation des degrés des polynômes en une variable.

2.4.1 L'espèce `algebre_formelle_univariee`

Nous allons donc définir une espèce de polynômes univariés à coefficients sur un anneau commutatif. On se rend bien compte que l'on utilise des opérations sur les naturels et la coercion avec les `int`, lors de la définition de la dérivée.

```
species algebre_formelle_univariee
      (r is anneau_commutatif, nat is naturels_ordonnes)
  inherits algebre_formelle_indexee(r,nat) =

  let derive =
    let pred(n) = non_failed (nat.predecesseur(n)) in
    let is_min(n) = nat.is_min(n) in
    let lc(p) = self.coefficient_dominant(p) in
    let reductum(p) = self.reductum(p) in
    let monome(coeff,deg) = self.monome(coeff,deg) in
    let degre(p) = self.degre(p) in
    let plus(x,y) = self.plus(x,y) in
    let mult(x,y) = self.module_multiplie(x,y) in
    let coeff_of_deg(d) = r.of_int(nat.to_int(d))
    in
    let rec aux(res, p) =
      let deg = self.degre(p) in
        if nat.is_min(deg) then
          res
        else
          aux(
            plus( mult((coeff_of_deg(deg)), (monome(lc(p), pred(deg)))),
                  res),
            reductum(p))
    in
      aux(self.zero);
;;
```

Les opérations entre les coefficients et les degrés nécessitent l'utilisation de la fonction `coeff_of_deg`. Cette fonction passe par les `int`, ce qui peut produire des erreurs à l'exécution si le degré du polynôme est trop important. Ceci n'est pas vraiment une limitation car cela permet de travailler avec des polynômes de degré 1073741823.

2.4.2 L'espèce `polynomes_corps_formels`

Nous définissons les polynômes univariés à coefficients sur un corps fini. Ces polynômes ont une structure d'anneau euclidien.

```
species polynomes_corps_formels
      (nat is semi_anneau_naturels,r is corps_fini(nat))
  inherits anneau_euclidien,
           algebre_formelle_univariee(r,nat) =

  sig fmecg in self -> r -> nat -> self -> self;
  let quo_rem(u,v) = ... ;
  let gcd_unitaire(x,y) = ... ;
  let random(n) = ... ;
;;
```

Donc, on implante la division euclidienne sur les polynômes. On ajoute également trois opérations :
- `fmecg(p,a,n,q)`, qui calcule $p + a * x^n q$,
- `gcd_unitaire(x,y)`, qui rend le pgcd unitaire de x et y,
- et `random(n)`, qui génère un polynôme de degré inférieur à `n`.

Le pgcd dans $\mathrm{K}[x]$ est défini à une constante non nulle près. On définit la méthode `gcd_unitaire` car il peut être pratique d'avoir toujours le même pgcd.

Chapitre 3

Les extensions algébriques

Il existait déjà dans la librairie la structure $\mathbb{Z}/n\mathbb{Z}$, mais mon but a été de généraliser à l'ensemble des structures quotients. Cela permet ainsi de construire facilement $\mathbb{Z}/n\mathbb{Z}$ ou K[x]/$\langle a(x) \rangle$.

3.1 Les structures quotients

3.1.1 Les groupes quotients

La structure quotient la plus simple est le groupe quotient. Pour le définir, il faut avoir un groupe et une relation d'équivalence.

Donc nous implantons le groupe quotient de la manière suivante :

```
species groupe_quotient (g is groupe_additif, reduire in (g -> g))
  inherits groupe_additif =

  rep T = g;

  let zero = g.zero;
  let is_zero = g.is_zero;
  let plus(x,y) = reduire(g.plus(x,y));
  let oppose(x) = reduire(g.oppose(x));
  let reduire(x) = reduire(x);
;;
```

On passe en paramètre un groupe, et la fonction `reduire` qui nous donne un représentant de la classe d'équivalence des éléments du groupe. Le groupe quotient a le même type support que le groupe `g`. Et on peut définir l'addition comme le représentant de la classe d'équivalence du résultat de l'addition dans le groupe.

3.1.2 Les anneaux quotients

Les anneaux quotients sont définis de la même façon que les groupes quotients, ils ont seulement la multiplication en plus.

```
species anneau_quotient (r is anneau_commutatif, reduire in (r -> r))
  inherits groupe_quotient(r,reduire), anneau_commutatif =

  let un = r.un;
  let is_un = r.is_un;
  let multiplie(x,y) = reduire(r.multiplie(x,y));
```

```
;;
```

Donc dans le cas général les anneaux quotients ont une structure d'anneau. Si on travaille avec un anneau euclidien A, on prend généralement la relation d'équivalence suivante :

$$(a,b) \in A,\ a \equiv b \Leftrightarrow n \mid b - a$$

où n est un élément de l'anneau. Dans ce cas, on peut prendre comme fonction qui donne un représentant de la classe d'équivalence, la fonction :

$$reduire(x) = x \bmod n.$$

L'espèce `anneau_euclidien_quotient`, utilise cette relation d'équivalence.

```
species anneau_euclidien_quotient (r is anneau_euclidien, n in r)
    inherits anneau_quotient(r, (fun x -> r.reste(x,n))) =
;;
```

Donc cette espèce est paramétrée par un anneau euclidien et un élément n dans cet anneau.

Dans certain cas, comme $\mathbb{Z}/n\mathbb{Z}$ avec n premier, les anneaux quotients ont une structure de corps. Et dans ce cas, on peut définir l'inverse.

```
species corps_quotient (r is anneau_euclidien, n in r)
  inherits anneau_euclidien_quotient(r,n), corps =

  let inverse(x) =
   let res = r.bezout(x,n)
   in
   let g = gcd(res) in
   let c1 = coef1(res) in
   if self.is_un(g)
   then c1
   else foc_error ("corps_quotient : inverse: Bad Gcd");
  let inverse_special(x) = self.inverse(x);
  let divise(x,y) = self.multiplie(x,(self.inverse_special(y)));
;;
```

La méthode `inverse_special` est là pour des raisons techniques ; nous allons voir ça dans ce qui suit.

3.2 Les entiers

Nous allons reconstruire $\mathbb{Z}/n\mathbb{Z}$ en utilisant les structures que nous venons de définir.

Dans le cas général nous avons :

```
species entiers_modulo (ent is entiers, the_mod in ent)
  inherits anneau_euclidien_quotient(ent, the_mod) =
;;
```

Il suffit d'hériter de l'espèce `anneau_euclidien_quotient` en lui passant en paramètre une collection d'anneau sur les entiers pour avoir toutes les propriétées.

Nous allons voir maintenant le cas où n est premier. Dans ce cas nous avons une structure de corps fini.

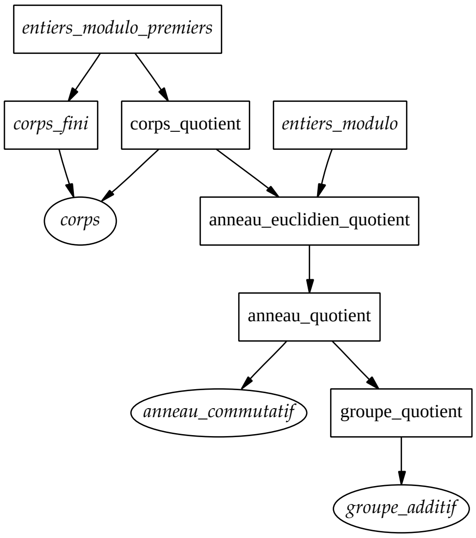
```
species entiers_modulo_premiers
  (ent is entiers, the_mod in ent, nat is semi_anneau_naturels)
```

```
    inherits corps_fini(nat), corps_quotient(ent,the_mod)
   =
    let cardinal = nat.of_int(ent.to_int(the_mod));
    let caracteristique = nat.of_int(ent.to_int(the_mod));
    let dimension = nat.un;

    let inverse_special(x) =
      let res = self.inverse(x) in
        if ent.gt(res,ent.zero)
        then res
        else self.plus(res,the_mod);
  ;;
```

Dans ce cas, on connait la caractéristique, qui est n, et la dimension qui est un, donc aussi le cardinal. Et nous sommes obligé de définir une méthode `inverse_special` car lors dans la définition de la méthode `inverse` dans l'espèce `corps_quotient`, il y a un appel à la méthode `bezout` de l'anneau euclidien passé en paramètre. Le résultat qui est rendu par cette méthode n'est pas nécessairement le représentant de la classe d'équivalence que nous voulons. Et c'est pour cela que l'on doit redéfinir la méthode inverse.

Finalement nous obtenons la hiérarchie suivante :



3.3 Les polynômes

De la même manière que nous venons de définir $\mathbb{Z}/n\mathbb{Z}$, nous allons définir $\mathbb{F}_q[x]/\langle a(x)\rangle$. Pour des raisons d'efficacité, nous supposons $a(x)$ unitaire.

```
species polynomes_modulo_unitaire
     (d is semi_anneau_naturels, r is corps_fini(d),
      ann_poly is polynomes_corps_formels(d*r), themod in ann_poly)
  inherits anneau_euclidien_quotient(ann_poly,themod) =

  let plus(x,y) = ann_poly.plus(x,y);

  let of_poly(x in ann_poly) in self = self.reduire(x);
  let to_poly(x in self) in ann_poly = x;
```

```
;;
```

En plus de l'héritage à `anneau_euclidien_quotient`, il y a la redéfinition de `plus` directement avec le plus de l'anneau, car l'addition ne fait pas augmenter le degré. Et il y a les méthodes `of_poly` et `to_poly` pour faire la coercion avec les polynômes.

Dans le cas, où le polynôme $a(x)$ est irréductible, nous avons l'espèce suivante :

```
species polynomes_modulo_irreductible
 (d is semi_anneau_naturels, r is corps_fini(d),
      ann_poly is polynomes_corps_formels(d*r), themod in ann_poly)
  inherits corps_fini(d),
          polynomes_modulo_unitaire(d,r,ann_poly,themod),
          corps_quotient(ann_poly,themod) =

  let caracteristique = r.caracteristique;
  let dimension = d.multiplie(r.dimension,ann_poly.degre(themod));
  let cardinal = d.expt(self.caracteristique, d.to_int(self.dimension));

  local let rand (x in int) in int = caml rand;
  let random(unit) = self.index(self.rand(d.to_int(self.cardinal)));

  let index =
    let card = d.to_int(self.cardinal) in
    let p = d.to_int(r.cardinal) in
    let nat_of_int(n) = d.of_int(n) in
    let coef_of_int(n) = r.of_int(n) in
    let monome(c,d) = ann_poly.monome(c,d) in
    let zero = self.zero in
    let plus(x,y) = self.plus(x,y)
    in
    let rec aux(i,j) =
      if int_gt(i,0) then
        let h = int_mod(i,p) in
          if int_eq(h,0) then
            aux(int_div(i,p),succ(j))
          else
            let a = coef_of_int(h) in
              plus(monome(a,nat_of_int(j)), aux(int_div(i,p),succ(j)))
      else
        zero
    in
      fun n ->
        aux(int_mod(n,card),0);

  let inverse(x) =
    let res = ann_poly.bezout(x,themod)
    in
    let g = gcd(res) in
    let c1 = coef1(res) in
    if self.is_un(g)
    then c1
    else
      if d.is_min(self.degre(g))
      then self.module_multiplie
```

```
        (r.inverse_special(self.coefficient_dominant(g)),c1)
      else foc_error ("polynomes_modulo_irreductible : inverse: Bad Gcd");

  let inverse_special(x) = self.inverse(x);
;;
```

La méthode `index`, permet de générer les éléments du corps fini à partir d'un entiers.

Chapitre 4

Factorisation de polynômes sur un corps fini

4.1 Introduction

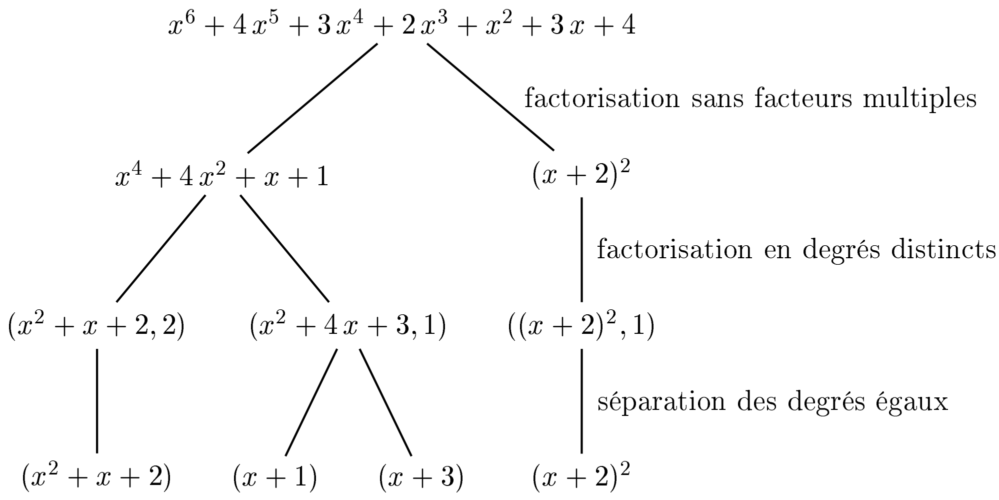
Nous allons commencer cette partie par la définition d'un polynôme factorisé et un exemple de factorisation, puis on vera en détail l'algorithme utilisé.

Définition 1
La décomposition sous forme de facteurs irréductibles de $a(x)$ est :

$$a(x) = \lambda \cdot \prod_{i=1}^{r} p_i(x)^{e_i}$$

où λ est une constante, les p_i sont unitaires et irréductibles, distincts deux à deux, et $e_i \in \mathbb{N}^*$.

La factorisation des polynômes se décompose en plusieurs étapes :
– la factorisation sans facteurs multiples ;
– la factorisation en degrés distincts ;
– la séparation des degrés égaux.
Voici un exemple de factorisation dans $\mathbb{F}_5$ (le corps fini a cinq éléments) :



le polynôme $a(x) = x^6 + 4\,x^5 + 3\,x^4 + 2\,x^3 + x^2 + 3\,x + 4$ est factorisé en deux polynômes sans facteurs multiples $x^4 + 4\,x^2 + x + 1$ et $x + 2$ dont le second est au carré.

Puis on applique la factorisation en degrés distincts sur tous les polynômes que nous a donné la décomposition sans facteurs multiples ; on a alors une décomposition en polynômes qui sont chacun produits de polynômes irréductibles de mêmes degrés. Par exemple la factorisation en degrés distincts du polynôme $x^4 + 4\,x^2 + x + 1$ donne deux polynômes ; le premier $x^2 + x + 2$ est produit de polynômes de degré 2 et le deuxième $x^2 + 4\,x + 3$ est produit de polynômes de degré 1.

Et enfin on applique la séparation des degrés égaux sur les polynômes ainsi trouvés. Ce qui nous donne la décomposition en facteurs irréductibles :

$$a(x) = (x^2 + x + 2) \cdot (x + 1) \cdot (x + 3) \cdot (x + 2)^2$$

4.2 La décomposition sans facteurs multiples

La décomposition sans facteurs multiples s'appelle aussi décomposition square-free. Nous allons différencier trois choses :

- un polynôme sans facteurs multiples ;
- la décomposition sans facteurs multiples d'un polynôme qui est l'écriture d'un polynôme sous forme de produit de polynômes sans facteurs multiples ;
- et la factorisation sans facteurs multiples d'un polynôme qui est l'algorithme qui donne la décomposition sans facteurs multiples.

Définition 2
Soit $a(x) \in K[x]$ un polynôme primitif à coefficients sur un corps K. $a(x)$ est un polynôme sans facteurs multiples s'il n'existe pas de $b(x)$ avec $degre(b(x)) \geq 1$ tel que

$$b(x)^2 \mid a(x).$$

Donc un polynôme sans facteurs multiples est de la forme :

$$a(x) = \prod_{i=1}^{r} p_i(x)$$

avec les $p_i(x)$ unitaires, irréductibles et distincts deux à deux.

Remarque :

Le fait qu'il n'y ait pas de facteur au carré est suffisant pour dire qu'il n'y a aucun facteur multiple. Car, par exemple, si on prend

$$a(x) = b(x)^3 \cdot c(x)$$

alors on peut le réécrire

$$a(x) = b(x)^2 \cdot (b(x) \cdot c(x)).$$

Définition 3
La décomposition sans facteurs multiples de $a(x)$ est :

$$a(x) = \lambda \cdot \prod_{i=1}^{k} a_i(x)^i$$

où $\lambda \in K$, et où chaque $a_i(x)$ est sans facteurs multiples ou égal à un et $a_i(x) \wedge a_j(x) = 1$ pour $i \neq j$ (le pgcd de $a_i(x)$ et $a_j(x)$ est égal à 1).

Exemple :

Si on prend le polynôme $a(x)$ sous sa forme complètement factorisée :

$$a(x) = p_1(x)^2 \cdot p_2(x)^3 \cdot p_3(x) \cdot p_4(x)^2 \cdot p_5(x)^5$$

Sa décomposition sans facteurs multiples est :

$$a(x) = (p_3(x))^1 \cdot (p_1(x) \cdot p_4(x))^2 \cdot (p_2(x))^3 \cdot (1)^4 \cdot (p_5(x))^5$$

Dans la suite de ce paragraphe nous allons voir comment trouver la décomposition sans facteurs multiples dans le cas où le polynôme est défini sur un anneau de caractéristique nulle puis dans le cas des corps finis.

4.2.1 La factorisation square-free dans un anneau de caractéristique nulle

Théorème 1

Soit $a(x) \in \mathsf{R}[x]$ un polynôme primitif et $a'(x)$ sa dérivée avec K un corps de caractéristique 0. $a(x)$ a un facteur multiple si et seulement si $a(x) \wedge a'(x) \neq 1$.

Démonstration :

On suppose que $a(x)$ a des facteurs multiples. On peut donc écrire :

$$a(x) = b^2(x) \cdot v(x)$$

On dérive $a(x)$:

$$a'(x) = 2 \cdot b(x) \cdot b'(x) \cdot v(x) + b^2(x) \cdot v'(x)$$

que l'on peut écrire sous la forme :

$$a'(x) = b(x).w(x)$$

On constate alors que $a(x) \wedge a'(x)$ n'est pas égal à 1. Donc si $a(x)$ a des facteurs multiples alors $a(x) \wedge a'(x) \neq 1$.

Il reste à démontrer que $a(x) \wedge a'(x) \neq 1$ implique $a(x)$ a des facteurs multiples. Nous allons démontrer cela par l'absurde. On suppose $a(x)$ un polynôme sans facteurs multiples et $c(x) = a(x) \wedge a'(x)$ différent de 1. Comme $a(x)$ est square-free, on sait que sa forme factorisée est la suivante :

$$a(x) = p_1(x) \cdot p_2(x) \cdots p_k(x)$$

avec p_i irréductible, $degre(p_i(x)) \geq 1$ et $p_i \wedge p_j = 1$ pour $i \neq j$. On dérive $a(x)$:

$$a'(x) = \sum_{i=1}^{k} \left(p_i'(x) \cdot \prod_{j \neq i} p_j(x) \right).$$

On suppose que $p_1(x)$ est un des facteurs communs à $a(x)$ et $a'(x)$ donc $p_1(x) \mid a'(x)$. Ceci implique :

$$p_1(x) \mid p_1'(x) \cdot p_2(x) \cdots p_k(x).$$

Comme tous les $p_i(x)$ sont premiers entre eux, on a :

$$p_1(x) \mid p_1'(x)$$

Comme $p_1'(x)$ est de degré inférieur à $p_1(x)$, on a nécessairement :

$$p_1'(x) = 0.$$

Comme $p_1(x)$ est à coefficients sur un anneau de caractéristique 0, alors $p_1(x)$ est constant ! On a une contradiction avec $degre(p_i(x)) \geq 1$, c'est à dire avec le fait que $a(x)$ soit sans facteurs multiples. Donc si $a(x) \wedge a'(x) \neq 1$ alors le polynôme $a(x)$ a des facteurs multiples.

Donc pour trouver la décomposition sans facteurs multiples de $a(x)$:

$$a(x) = \prod_{i=1}^{k} a_i(x)^i$$

il faut trouver ses facteurs communs avec sa dérivée

$$a'(x) = \sum_{i=1}^{k} a_1(x) \cdot a_2(x)^2 \cdots a_{i-1}(x)^{i-1} \cdot \left(i \cdot a_i(x)^{i-1} \cdot a_i'(x)\right) \cdot a_{i+1}(x)^{i+1} \cdots a_k(x)^k$$

donc

$$c(x) = a(x) \wedge a'(x) = \prod_{i=2}^{k} a_i(x)^{i-1}.$$

On calcule alors

$$w(x) = a(x)/c(x) = \prod_{i=1}^{k} a_i(x)$$

et

$$y(x) = c(x) \wedge w(x) = \prod_{i=2}^{k} a_i(x)$$

On a donc le premier facteur de la décomposition

$$a_1(x) = w(x)/y(x).$$

Puis, pour trouver les autres facteurs, on fait la même chose en repartant de $c(x)$. Lorsque $w(x)$ est égal à 1, alors on a trouvé tous les facteurs.

Ceci nous donne un algorithme simple pour trouver la décomposition sans facteurs multiples d'un polynôme $a(x)$ primitif à coefficients sur un anneau factoriel de caractéristique nulle. C'est la méthode de Musser.

```
square_free(a(x))
  i <- 1;
  res <- 1;
  b(x) <- a'(x);
  c(x) <- gcd(a(x),b(x));
  w(x) <- a(x)/c(x);

  TantQue w(x) ≠ 1 faire
    y(x) <- gcd(w(x),c(x));
    z(x) <- w(x)/y(x);
    res <- res.(z(x)^i);
```

```
      i <- i+1;
      w(x) <- y(x);
      c(x) <- c(x)/y(x);
    FinTantQue

    rendre(res);
  Fin
```

4.2.2 La factorisation square-free dans un corps fini

La méthode précédente ne permet pas la factorisation sur les corps finis $\mathbb{F}_q$ (q est le nombre d'éléments et $q = p^m$ avec p premier), car ils sont de caractéristique p non nulle. Le problème vient du fait que $a'(x) = 0$ pour des polynômes non constants.

Voici pour commencer un rappel de quelques propriétés et théorèmes sur les corps finis :

Théorème 2
Le petit théorème de Fermat, nous dit que pour tout élément r de $\mathbb{F}_q$ on a :

$$r^q = r$$

Lemme 1
$\forall\ s\,;\, r \in \mathbb{F}_q$ (q est le nombre d'éléments et p la caractéristique) on a :

$r^{1/p} = r^{q/p} = r^{p^{m-1}}$

$(r+s)^{p^j} = r^{p^j} + s^{p^j}$, $j = 0, 1..., m$

Le théorème suivant va nous permettre de résoudre le problème de la factorisation sans facteurs multiples dans les corps finis.

Théorème 3
Soit $a(x) = a_0 + a_1 x + \cdots + a_n x^n$ un polynôme de degré n de $\mathbb{F}_q$[x] avec $a'(x) = 0$.

Alors il existe $b(x)$ avec $a(x) = b(x)^p$.

Démonstration :

Si $a'(x) = 0$ alors les puissances non nulle de x dans $a(x)$ doivent être divisibles par p.
On peut écrire :

$$a(x) = a_0 + a_p x^p + a_{2p} x^{2p} + \cdots + a_{kp} x^{kp}.$$

On pose

$$b(x) = b_0 + b_1 x + \cdots + b_k x^k$$

où

$$b_i = a_{ip}^{1/p} = a_{ip}^{p^{m-1}}$$

et grâce au lemme on a :

$$\begin{aligned} b(x)^p &= b_0^p + b_1^p x^p + \cdots + b_k^p x^{kp} \\ &= a_0 + a_p x^p + \cdots + a_{kp} x^{kp} \\ &= a(x) \end{aligned}$$

On applique l'algorithme de Musser pour trouver la décomposition sans facteurs multiples de $a(x)$ à coefficients sur un corps fini de caractéristique p :

$$a(x) = \prod_{i=1}^{k} a_i(x)^i = \left(\prod_{p\nmid i} a_i(x)^i\right) \cdot \left(\prod_{p\mid i} a_i(x)^i\right)$$

on a écrit $a(x)$ sous forme d'un produit de polynôme sans facteurs multiples à la puissance i. Puis, on les a regroupés en polynômes qui sont à une puissance divisible par p ou non. Lorsque l'on dérive, les termes qui sont puissances de p s'annulent :

$$a'(x) = \sum_{p\nmid i} a_1(x) \cdot a_2(x)^2 \cdots a_{i-1}(x)^{i-1} \cdot \left(i \cdot a_i(x)^{i-1} \cdot a_i'(x)\right) \cdot a_{i+1}(x)^{i+1} \cdots a_k(x)^k$$

donc

$$c(x) = a(x) \wedge a'(x) = \left(\prod_{i\neq 1,\ p\nmid i} a_i(x)^{i-1}\right) \cdot \left(\prod_{p\mid i} a_i(x)^i\right).$$

On calcule alors

$$w(x) = a(x)/c(x) = \prod_{p\nmid i} a_i(x)$$

et

$$y(x) = c(x) \wedge w(x) = \prod_{i\neq 1,\ p\nmid i} a_i(x)$$

On constate ici deux problèmes :
- $a'(x)$ peut être nul pour $a(x)$ non constant ;
- $w(x)$ est produit des facteurs qui ne sont pas multiples de p. Donc lorsque $w(x) = 1$, il reste tous les facteurs $\prod_{p\mid i} a_i(x)^i$.

Le théorème précédent nous permet de résoudre ces problèmes. On peut donc modifier l'algorithme précédent pour avoir la factorisation sans facteurs multiples sur un corps fini. `a(x)` est un polynôme unitaire (son coefficient dominant est un) et `p` est la caractéristique du corps.

```
square_free(a(x))
  i <- 1;
  res <- 1;
  b(x) <- a'(x);

  Si b(x) ≠ 0 alors
    c(x) <- gcd(a(x),b(x));
    w(x) <- a(x)/c(x);

    TantQue w(x) ≠ 1 faire
      y(x) <- gcd(w(x),c(x));
      z(x) <- w(x)/y(x);
      res <- res.(z(x)^i);
      i <- i+1;
      w(x) <- y(x);
      c(x) <- c(x)/y(x);
    FinTantQue
    Si c(x) ≠ 1 alors
      c(x) <- c(x)^(1/p);
```

```
      res <- res.(square_free(c(x)))^p;
    FinSi
  Sinon
    a(x) <- a(x)^(1/p);
    res <- res.(square_free(a(x)))^p;
  FinSi
  rendre(res);
Fin
```

Exemple :

On applique l'algorithme dans $\mathbb{F}_2$ pour trouver la décomposition sans facteurs multiples du polynôme

$$a(x) = x^{24} + x^{20} + x^{14} + x^8 + x^2 + 1.$$

La dérivée de $a(x)$ est nulle. Il faut donc calculer la décomposition de

$$a(x)^{1/2} = x^{12} + x^{10} + x^7 + x^4 + x + 1.$$

La dérivée de ce polynôme est $x^6 + 1$. On calcule le pgcd du polynôme avec sa dérivée :

$$c(x) = x^6 + 1$$

et

$$w(x) = x^6 + x^4 + x + 1$$

On entre dans la boucle :

$$y(x) = x^3 + 1$$

Et donc le premier facteur trouvé est :

$$z(x) = w(x)/y(x) = x^3 + x + 1$$

Le prochain facteur sera au carré. Et on refait le calcul avec

$$w(x) = x^3 + 1$$
$$c(x) = x^3 + 1.$$

Ceci nous donne comme facteur au carré $x^3 + 1$. Puis on sort de la boucle, et on doit alors prendre le résultat au carré. Donc la décomposition de $a(x)$ est :

$$a(x) = (x^3 + x + 1)^2 \cdot (x^3 + 1)^6.$$

4.2.3 L'implantation

Nous allons commencer par voir comment on exprime le théorème 3, puis une implantation de la méthode de Musser pour la factorisation sans facteurs multiples.

Pendant mon stage, j'ai seulement énoncé les théorèmes, je n'ai pas fait leur démonstration.

```
theorem deriv_0 =
  all a in self,
   (self.is_zero(self.derive(a)) ->
      (
        ex b in self,
          self.equal(a, self.expt(b,nat.to_int(r.caracteristique)))
      )
   )
  proof: axiom;
```

L'algorithme que nous avons écrit est pour la factorisation sans facteurs multiples d'un polynôme univarié, à coefficients sur un corps fini de caractéristique `carac` et dont les degrés sont des entiers. Le résultat est rendu sous forme d'une liste de couple (`poly`, `puiss`) où `poly` est un polynôme sans facteurs multiples et `puiss` la puissance où se trouve `poly` dans le polynôme à factoriser.

```
let sqrfree(a) =
  let rec boucle(cpt, w, c, res) =
    if is_un(w) then
      crp(c,res)
    else
      let y = gcd(w,c) in
      let z = sdiv(w,y) in
        if is_un(z) then
          boucle(succ(cpt), y, sdiv(c,y), res)
        else
          boucle(succ(cpt), y, sdiv(c,y), Cons(crp(z,cpt),res))
  in

  let aux(x) = let c = first(x) in let d = scnd(x) in crp(c,nmult(d,carac))
   in

  let rec sqf(a) =
    let b =derive(a) in
      if is_zero(b) then
        let da = degre(a) in
          if is_min(da) then
            Cons(crp(a,nat_un),Nil)
          else
            let new_a = puiss_1_p(a) in
            let dec = sqf(new_a) in
              map(aux, dec)
      else
        let c = gcd(a,b) in
        let w = sdiv(a,c) in
        let couple_res = boucle(nat_un, w, c, Nil) in
        let new_c = first(couple_res) in
        let res = scnd(couple_res) in
          if is_un(new_c) then
            res
          else
            let csp = puiss_1_p(new_c) in
            let dec = sqf(csp) in
              append(res,(map(aux, dec)))
  in
    let coef_dom =lc(a) in
      if coef_is_un(coef_dom) then
        sqf(a)
      else
        if is_zero(a) then
          Cons(crp(a,nat_un),Nil)
        else
          Cons(crp(monome(coef_dom,nat_zero),nat_un),
               sqf(mmult(inverse(coef_dom), a)));
```

Nous utilisons les fonctions suivantes :
- les opérations sur les polynômes :
  - `is_un(p)`, teste si le polynôme `p` est égal à un,
  - `is_zero(p)`, teste si le polynôme `p` est égal à zéro,
  - `gcd(x,y)`, calcule le pgcd des polynômes x et y,
  - `degre(p)`, donne le degré de `p`,
  - `derive(p)`, calcule la dérivée de `p`,
  - `sdiv(x,y)`, fait la division exacte de `x` par `y`,
  - `mmult(cte,p)`, multiplie le polynôme `p` par la constante `cte` (la constante est du même type que les coefficients),
  - `monome(coef,deg)`, crée le polynôme $\texttt{coef} \cdot x^{\texttt{deg}}$,
  - `lc(p)`, donne le coefficient dominant de `p`,
  - `zero`, le polynôme 0,
  - `puiss_1_p(poly,p)`, calcule $\texttt{poly}^{1/\texttt{p}}$ où `p` est la caractéristique ;
- les opérations sur les entiers :
  - `succ(n)`, donne le successeur de `n`,
  - `is_min(n)`, teste si `n` est égal à zéro,
  - `nmult(m,n)`, la multiplication des entiers,
  - `nat_un`, l'entier 1,
  - `nat_zero`, l'entier 0 ;
- les opérations sur les coefficients :
  - `coef_is_un coef`, teste si l'élément `coef` est l'unité du corps fini,
  - `inverse(coef)`, rend l'inverse de `coef` ;

La première chose qui est faite lors de l'appel de la méthode `sqrfree` est de rendre le polynôme unitaire. Puis la fonction `sqf` est appelée. Elle teste si le polynôme est sans facteurs multiples et appelle la fonction `boucle` s'il ne l'est pas. La fonction `boucle` calcule des facteurs de la décomposition. Au retour de la fonction `boucle`, la fonction `sqf` teste s'il reste des facteurs à calculer et alors soit elle se rappelle récursivement soit elle rend le résultat.

On constate que l'on doit faire des opérations entre la caractéristique et les degrés des polynômes, c'est pour cela qu'il faut faire attention que la collection dans laquelle il sont définis soit la même.

4.3 La décomposition en degrés distincts

Nous allons étudier ici la seconde partie de la factorisation. On se place toujours sur les corps finis et on travaille avec des polynômes sans facteurs multiples.

4.3.1 L'algorithme

Définition 4
La décomposition en degrés distincts d'un polynôme unitaire sans facteurs multiples $a(x)$ à coefficients sur un corps fini est :

$$a(x) = \prod_i a_i(x)$$

où chaque $a_i(x)$ est le produit de polynômes irréductibles de degré i.

Exemple :

On écrit le polynôme sans facteurs multiples $a(x)$ sous forme complètement factorisée :

$$a(x) = \prod_{i=1}^{r} p_{i,d_i}(x)$$

où les $p_{i,d_i}(x)$ sont des polynômes unitaires irréductibles de degré d_i et deux à deux distincts. Alors, par exemple, la décomposition en degrés distincts de

$$a(x) = p_{1,3}(x) \cdot p_{2,2}(x) \cdot p_{3,2}(x) \cdot p_{4,1}(x) \cdot p_{5,3}(x)$$

est

$$a(x) = (p_{4,1}(x)) \cdot (p_{2,2}(x) \cdot p_{3,2}(x)) \cdot (p_{1,3}(x) \cdot p_{5,3}(x)).$$

Théorème 4
Le polynôme $p_r(x) = x^{q^r} - x$ est le produit de tous les polynômes irréductibles, unitaires, à coefficients sur $\mathbb{F}_q$ de degré qui divise r

Démonstration :

Dans un premier temps, nous allons voir que le polynôme $p_r(x)$ est sans facteurs multiples. Sa dérivée est

$$\begin{aligned} p_r(x)' &= q^r \cdot x^{q^r - 1} - 1 \\ &= -1 \end{aligned}$$

car on est sur $\mathbb{F}_q[x]$. Le pgcd de $p_r(x)$ avec sa dérivée est 1, donc il est sans facteurs multiples.
On montre maintenant que tous les polynômes irréductibles, unitaires, à coefficients sur $\mathbb{F}_q$ de degré qui divise r sont facteurs de $p_r(x)$. Soit $m(x)$ un polynôme irréductible de degré d de $\mathbb{F}_q[x]$ avec $d \mid r$.

$$\mathrm{F} = \mathbb{F}_q[x]/\langle m(x) \rangle$$

Comme $m(x)$ est irréductible, F est un corps. De plus comme $m(x)$ est de degré d, alors F a q^d éléments.
D'après le petit théorème de Fermat on a :

$$f^{q^d} = f$$

et aussi

$$f^{q^r} = ((f^{q^d})^{q^d} \cdots)^{q^d} = f.$$

Donc si on considère $f = [x]$ la notation du polynôme x dans F, d'après l'équation précédente on a :

$$[p_r(x)] = [x^{q^r} - x] = [x]^{q^r} - [x] = f^{q^r} - f = 0$$

dans F. Ce qui est équivalent a :

$$p_r(x) \equiv 0 \bmod m(x)$$

Donc $m(x)$ divise $p_r(x)$.

Enfin, pour montrer par l'absurde que ce n'est pas le produit d'autres polynômes, on suppose $m(x)$ irréductible de degré d qui divise $p_r(x)$ avec $d > r$. Donc dans le corps F on a :

$$[x]^{q^r} - [x] = [p_r(x)] = 0$$

On prend un élément de F

$$\begin{aligned} f &= [f_0 + f_1 \cdot x + \cdots + f_{d-1} \cdot x^{d-1}] \\ &= f_0 + f_1 \cdot [x] + \cdots + f_{d-1} \cdot [x]^{d-1} \end{aligned}$$

avec $f_i \in \mathbb{F}_q$.
Grâce à l'équation précédente on a

$$\begin{aligned} f^{q^r} &= (f_0 + f_1 \cdot [x] + \cdots + f_{d-1} \cdot [x]^{d-1})^{q^r} \\ &= f_0^{q^r} + f_1^{q^r} \cdot [x]^{q^r} + \cdots + f_{d-1}^{q^r} \cdot [x]^{q^r \cdot d-1} \\ &= f_0^{q^r} + f_1^{q^r} \cdot [x] + \cdots + f_{d-1}^{q^r} \cdot [x]^{d-1} \\ &= f_0 + f_1 \cdot [x] + \cdots + f_{d-1} \cdot [x]^{d-1} \\ &= f \end{aligned}$$

Mais tous les éléments de F sont racines du polynôme $p_r(x)$. Comme il y a plus d'éléments dans F que le degré de $p_r(x)$, on a bien une contradiction.

Donc voici une méthode pour trouver la décomposition en degrés distincts. On calcule

$$a_1(x) = a(x) \wedge (x^q - x)$$

ce qui nous donne le produit de tous les facteurs de degré un. On simplifie $a(x)$:

$$a(x) = a(x)/a_1(x)$$

Puis on trouve les facteurs de degré 2 :

$$a_2(x) = a(x) \wedge (x^{q^2} - x)$$

On continue jusqu'à $a_{n/2}$ (n degré de $a(x)$) car après, le prochain facteur est nécessairement de degré n.

On a alors un algorithme pour calculer la décomposition en polynômes produits de facteurs irréductibles de même de degré d'un polynôme `a(x)` dans $\mathbb{F}_q[x]$:

```
Distinct_Degree(a(x))
  i <- 1;
  w(x) <- x;
  a0(x) <- 1;

  TantQue i <= degre(a(x))/2 Faire
    w(x) <- w(x)^q mod a(x);
    ai(x) <- gcd(a(x),w(x)-x);
    Si ai(x) ≠ 1 Alors
      a(x) <- a(x)/ai(x);
      w(x) <- w(x) mod a(x);
    FinSI
```

```
    i <- i+1;
  FinTantQue

  rendre(a0(x)...ai-1(x)a(x));
Fin
```

Exemple :

Nous allons appliquer l'algorithme dans $\mathbb{F}_2[x]$ sur le polynôme

$$a(x) = x^{15} + 1$$

On pose $i = 1$, $w(x) = x$ et $a_0 = 1$. Et on rentre dans la boucle.

$$w(x) = x^2$$

et donc le premier facteur est

$$a_1(x) = a(x) \wedge (x^2 - x) = x + 1$$

Puis pour avoir le second, on simplifie $a(x)$:

$$a(x)/a_1(x) = x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Et on recommence le calcul :

$$w(x) = x^{2^2} = x^4$$

et le facteur qui est composé de facteurs irréductibles de degré 2 est :

$$a_2(x) = a(x) \wedge (x^4 - x) = x^2 + x + 1$$

On continue et on trouve la décomposition :

$$\begin{aligned} a_1(x) &= x + 1 \\ a_2(x) &= x^2 + x + 1 \\ a_4(x) &= x^{12} + x^9 + x^6 + x^3 + 1 \end{aligned}$$

On peut donc dire que $a_1(x)$ et $a_2(x)$ sont des facteurs irréductibles de $a(x)$ et que $a_4(x)$ est composé de trois facteurs irréductibles de degré 4 (car $3 * 4 = 12$).

4.3.2 L'implantation

Le théorème que nous avons énoncé pour cette étape de la factorisation est le théorème 4. Pour pouvoir l'exprimer simplement en FOC, il a été décomposé en deux parties.

On exprime ici que si un polynôme m est unitaire irréductible de degré qui divise n, alors le polynôme $p = x^{q^n} - x$ est multiple de m.

```
theorem prod_all_irreductible1 =
  all n in nat,
    (ex p in self,
      (self.equal(p,self.moins(
                 self.monome(r.un,nat.expt(r.cardinal,nat.to_int(n))),
                 self.monome(r.un,nat.un))) and
        all m in self,
```

```
          (r.is_un(self.coefficient_dominant(m)) ->
            (nat.is_multiple(n,self.degre(m)) ->
              (self.is_irreductible(m) ->
                 self.is_multiple(p,m)
              )
            )
          )
      )
    )
  proof: axiom;
```

Dans l'autre partie du théorème, on dit que si m est de degré qui ne divise pas n, alors p n'est pas multiple de m.

```
theorem prod_all_irreductible2 =
  all n in nat,
    (ex p in self,
      (self.equal(p,self.moins(
                 self.monome(r.un,nat.expt(r.cardinal,nat.to_int(n))),
                 self.monome(r.un,nat.un))) and
        all m in self,
          (r.is_un(self.coefficient_dominant(m)) ->
            (not(nat.is_multiple(n,self.degre(m))) ->
              (self.is_irreductible(m) ->
                 not(self.is_multiple(p,m))
              )
            )
          )
      )
    )
  proof: axiom;
```

Nous voyons maintenant l'implantation de l'algorithme. Cette méthode prend en entrée un polynôme sans facteurs multiples et rend une liste de couples (`poly`,`deg`) où `poly` est le produit de polynômes irréductibles de degré `deg`.

```
let distinct_degree(a) =
  let rec boucle(i, a, w, res) =
    let i_fois_2 = dmult(i,deux) in
      if gt(i_fois_2,degre(a)) then
        if is_un(a) then
          match res with
            | Nil -> Cons(crp(a,zero),Nil)
            | _ -> res
        else
          Cons(crp(a,degre(a)),res)
      else
        let nouv_w = modulo(expt(w,card),a) in
        let ai = gcd(a, moins(nouv_w,poly_x)) in
          if is_un(ai) then
            boucle(succ(i), a, nouv_w, res)
          else
            let nouv_a = div(a,ai) in
              boucle(succ(i), nouv_a,
                     modulo(nouv_w,nouv_a), Cons(crp(ai,i),res))
  in
```

```
boucle(un, a, poly_x, Nil);
```

les fonctions utilisées sont les suivantes :
- les opérations sur les naturels :
  - `dmult(m,n)`, la multiplication dans les naturels,
  - `gt(m,n)`, teste si `m` est strictement supérieur à `n`,
  - `succ(n)`, calcule $\mathtt{n}+1$,
  - `deux`, est l'entier 2,
  - `un`, est l'entier 1,
  - `zero`, est l'entier 0 ;
- les opérations sur les polynômes :
  - `degre(p)`, donne le degré de `p`,
  - `expt(p,n)`, élève le polynôme `p` à la puissance `n` (si `n` est une puissance de la caractéristique),
  - `moins(x,y)`, la soustraction de deux polynômes,
  - `div(x,y)`, la division exacte de deux polynômes,
  - `modulo(x,y)`, calcule `x` modulo `y`,
  - `gcd(x,y)`, calcule le pgcd des polynômes `x` et `y`,
  - `is_un(p)`, teste si le polynôme est égal à un,
  - `poly_x`, le polynôme x,
  - `poly_un`, le polynôme 1.

La partie qui est coûteuse en temps est le calcul de `nouv_w`. Ce calcul consiste à élever le polynôme `w` à la puissance `card` puis de prendre le reste de la division euclidienne avec `a`. Comme la puissance à laquelle on élève le polynôme est le cardinal du corps, ce calcul peut ce faire très facilement grâce au petit théorème de Fermat. Mais après, le calcul du modulo est long. Une autre méthode est de calculer la puissance avec une dichotomie, mais en prenant le modulo avec `a` à chaque étapes. La seconde méthode est plus performante. Mais nous allons voir, dans le chapitre sur les optimisations, comment l'itérateur de Frobenius peut nous permettre d'avoir de meilleurs performances.

4.4 La séparation des degrés égaux

4.4.1 L'algorithme

Après la factorisation en degrés distincts, nous avons des polynômes qui sont produits de polynômes irréductibles de même degré. Nous voulons pour chacun des polynômes trouver les facteurs irréductibles dont ils sont composés.

Pour cette étape, nous devons appliquer des algorithmes différents suivant la parité de la caractéristique du corps.

Cas de la caractéristique impaire

Nous allons étudier l'algorithme de Cantor-Zassenhaus pour cette étape. Cet algorithme est correct uniquement sur $\mathbb{F}_q[x]$ avec q puissance d'un nombre premier impaire. Nous voulons factoriser le polynôme $a(x) \in \mathbb{F}_q[x]$ de degré n. Et nous avons un diviseur d de n tel que tous les facteurs irréductibles de $a(x)$ sont de degré d. Le nombre de facteurs irréductibles est $r = n/d$. On écrit les facteurs irréductibles de $a(x)$, $a_i(x)$ avec $i = 1, ..., r$ donc

$$a(x) = \prod_{i=1}^{r} a_i(x)$$

Si $r = 1$ alors $a(x)$ est irréductible. Sinon on a $a_i(x) \wedge a_j(x) = 1$ pour $i \neq j$. Donc grâce au théorème du reste chinois, nous avons :

$$\chi : R = \mathbb{F}_q[x]/\langle a(x)\rangle \to \mathbb{F}_q[x]/\langle a_1(x)\rangle \times \cdots \times \mathbb{F}_q[x]/\langle a_r(x)\rangle = R_1 \times \cdots \times R_r$$

où chaque R_i est un corps a q^d éléments :

$$\mathbb{F}_{q^d} \cong R_i = \mathbb{F}_q[x]/\langle a_i(x)\rangle \supseteq \mathbb{F}_q$$

Pour $f(x) \in \mathbb{F}_q[x]/\langle a(x)\rangle$, on a

$$(f(x))^{q^d} - f(x) = 0$$

comme nous l'avons vu dans la démonstration du théorème 4. De plus, comme la caractéristique est impaire, cette équation se décompose en

$$f(x)^{q^d} - f(x) = f(x)(f(x)^{\frac{q^d-1}{2}} - 1)(f(x)^{\frac{q^d-1}{2}} + 1) = 0.$$

D'autre part, on pose $e = \frac{q^d-1}{2}$, comme $a(x)$ est sans facteurs multiples, cela entraîne que $(f(x)^e \wedge a(x)) = (f(x) \wedge a(x))$. De plus $f(x)^e$, $f(x)^e - 1$ et $f(x)^e + 1$ sont premiers entre eux dans $\mathbb{F}_q[x]$ (car la caractéristique est différente de 2). Il en est donc de même pour $f(x)$, $f(x)^e - 1$ et $f(x)^e + 1$. Par conséquent, $f(x)(f(x)^e - 1)(f(x)^e + 1) = 0$ implique :

$$\gcd(f(x), a(x)) \times \gcd(f(x)^e - 1, a(x)) \times \gcd(f(x)^e + 1, a(x)) = a(x).$$

De là, on peut tirer un premier algorithme :

```
split_degree(a(x),d)
  n <- degre(a(x));
  r <- n/d;
  facteurs <- {a(x)};
  TantQue SizeOf(facteurs) < r Faire
    f(x) <- RandPoly(n);
    g(x) <- gcd(f(x),a(x));
    Si g(x) ≠ 1 Alors
      facteurs <- split_degree(g(x),d) ∪ split_degree(a(x)/g(x),d);
    Sinon
      g(x) <- gcd(f(x)^((q^d-1)/2),a(x));
      Si (g(x) ≠ 1) ET (g(x) ≠ a(x)) Alors
        facteurs <- split_degree(g(x),d) ∪ split_degree(a(x)/g(x),d);
      FinSi
    FinSi
  FinTantQue
Fin
```

Mais la question est de savoir s'il est rare ou non de tirer un élément qui convient. Nous utilisons la convention que pour $f(x) \in \mathbb{F}_q[x]$, on a

$$f(x) \bmod a(x) \in R$$

et

$$\chi(f(x)) = (f(x) \bmod a_1(x), ..., f(x) \bmod a_r(x)) = (\chi_1(f(x)), ..., \chi_r(f(x)))$$

où $\chi_i(f(x)) = f(x) \bmod a_i(x) \in R_i$. Pour $f(x) \in \mathbb{F}_q[x]$, $a_i(x)$ divise $f(x)$ si et seulement si $\chi_i(f(x)) = 0$. Si on a un $f(x)$ avec certain $\chi_i(f(x))$ nuls et d'autres non nuls alors $f(x) \wedge a(x)$ est un diviseur non trivial de $a(x)$.

Si on choisit aléatoirement $f(x) \in \mathbb{F}_q[x]$ de degré inférieur à n et avec $f(x) \wedge a(x) = 1$, on a alors $\gcd(f(x)^e - 1, a(x)) \times \gcd(f(x)^e + 1, a(x)) = a(x)$. Donc pour trouver un diviseur non trivial de $a(x)$, il faut que $\gcd(f(x)^e - 1, a(x))$ soit différent de 1 et de $a(x)$. Autrement dit, $f(x)$ ne convient pas si

$$\chi(f(x)^e - 1) = 0 \text{ ou } \chi(f(x)^e + 1) = 0$$

donc, par l'isomorphisme, il faut pour tout i :

$$\chi_i(f(x)^e - 1) = 0 \text{ ou } \chi_i(f(x)^e + 1) = 0.$$

Comme $\chi_i(f(x)^e) = \pm 1$ avec la même probabilité. Et qu'il faut que les r (nombre de termes) $\chi_i(f(x)^e)$ soient tous égaux à 1 ou tous égaux à -1. Alors la probabilité de trouver un élément qui ne convient pas est :

$$Prob(d\acute{e}favorable) = 2 \cdot (1/2)^r = (1/2)^{r-1}.$$

Donc la probabilité que le choix du polynôme soit bon est supérieur ou égal à 1/2.

Cas de la caractéristique 2

Nous nous plaçons dans le cas où le polynôme est à coefficients sur un corps fini de caractéristique 2.

Dans ce cas, on a toujours :

$$f(x)^{q^d} - f(x) = 0$$

mais la décomposition

$$f(x)^{q^d} - f(x) = f(x)(f(x)^{\frac{q^d-1}{2}} - 1)(f(x)^{\frac{q^d-1}{2}} + 1)$$

ne marche plus. Pour trouver une autre factorisation de ce polynôme, nous devons définir la fonction trace.

Définition 5
Soit F un corps fini de cardinalité q et K une extension de F de degré n. Alors, la fonction trace est défini sur K relativement à F par

$$Tr(x) = \sum_{i=0}^{n-1} x^{q^i}$$

Ce qui nous donne dans le cas où $F = \mathbb{F}_2$ et $K = \mathbb{F}_{2^m}$:

Définition 6
Le polynôme trace est défini sur $\mathbb{F}_{2^m}$ par

$$T_m(x) = x + x^2 + x^4 + \cdots + x^{2^{m-2}} + x^{2^{m-1}}.$$

Lemme 2
Le polynôme trace $T_m(x)$ défini sur $\mathbb{F}_{2^m}$ a les propriétés suivantes :
- $\forall v; w \in \mathbb{F}_{2^m}, T_m(v + w) = T_m(v) + T_m(w)$;
- $\forall v \in \mathbb{F}_{2^m}, T_m(v) \in \mathbb{F}_2$;
- $x^{2^m} - x = T_m(x) \cdot (T_m(x) + 1)$

Démonstration :

Pour la première propriété on a :

$$\begin{aligned}
T_m(v+w) &= (v+w)+(v+w)^2+\cdots+(v+w)^{2^{m-1}} \\
&= (v+w)+(v^2+w^2)+\cdots+(v^{2^{m-1}}+w^{2^{m-1}}) \text{ (car le corps est de caractéristique 2)} \\
&= (v+v^2+\cdots+v^{2^{m-1}})+(w+w^2+\cdots+w^{2^{m-1}}) \\
&= T_m(v)+T_m(w)
\end{aligned}$$

Pour la seconde propriété on a :

$$\begin{aligned}
T_m(v)^2 &= (v+v^2+\cdots+v^{2^{m-1}})^2 \\
&= v^2+v^4+\cdots+v^{2^{m-1}}+v^{2^m} \\
&= v^2+v^4+\cdots+v^{2^{m-1}}+v \\
&= T_m(v)
\end{aligned}$$

Donc, comme $\forall v \in \mathbb{F}_{2^m}$, $T_m(v)$ est racine du polynôme x^2-x. De plus $\mathbb{F}_2$ donne deux racines de ce polynôme dans $\mathbb{F}_{2^m}$, donc $T_m(v)$ est l'une d'elle.

Pour la troisième propriété, on prend α un élément de $\mathbb{F}_{2^m}$. Grâce à la propriété précédente on $T_m(\alpha)$ est égal à 0 ou 1. Donc α est racine de $T_m(x)\cdot(T_m(x)+1)$. Et donc

$$x^{2^m}-x=\prod_{\alpha\in\mathbb{F}_{2^m}}(x-\alpha) \text{ divise } T_m(x)\cdot(T_m(x)+1)$$

De plus $T_m(x)$ et $T_m(x)+1$ sont de degré 2^{m-1}, et le degré de $T_m(x)\cdot(T_m(x)+1)$ est 2^m. Donc $T_m(x)\cdot(T_m(x)+1)$ et $x^{2^m}-x$ ont le même degré et le second est un divise le premier donc $T_m(x)\cdot(T_m(x)+1)=x^{2^m}-x$.

On pose $q=2^m$, $a\in\mathbb{F}_q[x]$ un polynôme sans facteurs multiples, composé de r polynômes irréductibles $a_1,\ldots,a_r\in\mathbb{F}_q[x]$ de degré d. On définit R, R_i et χ_i comme précédemment. Grâce au lemme, on a :

$$x^{q^d}-x=x^{2^{m\cdot d}}-x=T_{m\cdot d}(x)\cdot(T_{m\cdot d}(x)+1)$$

donc $\forall\alpha\in R$ on a $\chi_i(T_{m\cdot d}(\alpha))\in\mathbb{F}_2$. Et donc on se retrouve comme dans le cas où la caractéristique est impaire.

On peut donc définir un algorithme général pour cette étape de la factorisation.

```
split_degree(a(x),d)
  n <- degre(a(x));
  facteurs <- {};
  aux <- {a(x)};
  TantQue aux ≠ {} Faire
    f(x) <- RandPoly(n);
    Si q = 2 Alors
      f(x) <- trace2mod(f(x),a(x));
    Sinon
      f(x) <- gcd(f(x)^((q^d-1)/2),a(x));
    FinSi
    nouv_aux <- {};
    Pour u(x) Dans aux Faire
      g(x) <- gcd(f(x),u(x));
      Si (g(x) ≠ 1) ET (g(x) ≠ a(x)) Alors
```

```
        Enlever(u(x),aux);
        Si degre(g(x)) = d Alors
          facteurs <- facteurs ∪ {g(x)};
        Sinon
          nouv_aux <- nouv_aux ∪ {g(x)};
        FinSi
        Si degre(u(x)/g(x)) = d Alors
          facteurs <- facteurs ∪ {u(x)/g(x)};
        Sinon
          nouv_aux <- nouv_aux ∪ {u(x)/g(x)};
        FinSi
      FinSi
    FinPour
    aux <- nouv_aux;
  FinTantQue
Fin
```

4.4.2 L'implantation

Cette implantation de la séparation des degrés égaux regroupe les deux cas que nous avons vu précédemment, il y a un test dans le code qui permet d'appliquer le bon algorithme. Cette méthode prend en paramètre un polynôme `a` sans facteurs multiples qui est produit de polynômes irréductibles de même degré `n`. Le résultat de la méthode est la liste des facteurs irréductibles de `a`.

```
let split_degree(a,n) =
  let rec boucle(aux, nouv_aux, f, d, res) =
    match aux with
      | Nil -> crp(nouv_aux,res)
      | Cons(u,suite) -> (
             let g = gcd(u,f) in
             let dg = degre(g) in
               if or_b(egal(dg,zero),egal(dg,degre(u)))
               then boucle(suite, Cons(u,nouv_aux), f, d, res)
               else (
                 let v = div(u,g) in
                 let tmp =
                   if egal(dg,d)
                   then crp(Cons(g,res),nouv_aux)
                   else crp(res,Cons(g,nouv_aux))
                 in
                 let nres = first(tmp) in
                 let nnaux = scnd(tmp) in
                   if egal(degre(v),d)
                   then boucle(suite, nnaux, f, d, Cons(v,nres))
                   else boucle(suite, Cons(v,nnaux), f, d, nres)
               )
           )
  in

  let rec equal_deg_split(carac_paire, a, f, aux, res) =
    match aux with
      | Nil -> res
      | aux -> (
```

```
                  let t = random(mult(deux,d)) in
                  let f =
                    if carac_paire
                    then trace2mod(t, mult(pred(cps_dim),pred(d)), a)
                    else (
                      moins(
                        expt_modulo(t,ndiv(pred(nexpt(card,d)),deux),a),
                        un)
                    )
                  in
                  let tmp = boucle(aux, Nil, f, d, res) in
                  let naux = first(tmp) in
                  let nres = scnd(tmp) in
                    equal_deg_split(carac_paire, a, d, naux, nres)
                )
      in
        let carac_paire = egal(carac, deux) in
        let deg = degre(a) in
          if leq(deg,d) then
            Cons(a,Nil)
          else (
            equal_deg_split(carac_paire, a, d, Cons(a,Nil), Nil)
          );
```

- les opérations sur les polynômes :
  - `degre(p)`, donne le degré de `p`,
  - `moins(x,y)`, la soustraction de polynôme,
  - `div(x,y)`, fait le division de `x` par `y`,
  - `un`, le polynôme 1,
  - `gcd(x,y)`, calcule le pgcd de `x` et `y`,
  - `is_un(p)`, teste si `p` est égal à un,
  - `random(n)`, génère un polynôme de degré inférieur à `n`,
  - `trace2mod(v,m,themod)`, calcule le polynôme trace $T_{\mathtt{m}}(\mathtt{v})$, modulo le polynôme `themod`,
  - `expt_modulo(poly,puiss,themod)`, calcule $\mathtt{poly}^{\mathtt{puiss}} \bmod \mathtt{themod}$;
- les opérations sur les entiers :
  - `leq(m,n)`, teste si `m` est inférieur ou égal à `n`,
  - `egal(m,n)`, teste si `m` est égal à `n`,
  - `ndiv(m,n)`, division exacte sur les entiers,
  - `mult(m,n)`, la multiplication,
  - `nexpt(m,n)`, élève `m` à la puissance `n`,
  - `deux`, l'entier 2,
  - `zero`, l'entier 0,
  - `pred(n)`, calcule $\mathtt{n} - 1$;
- les informations sur le corps :
  - `carac`, la caractéristique du corps,
  - `cps_dim`, la dimension du corps ;

Dans la méthode `split_degree`, la fonction `split` test si le polynôme est irréductible, puis s'il ne l'est pas appelle la fonction `equal_deg_split` pour trouver un des facteurs. Et une fois ce facteur trouvé se rappelle récursivement avec en paramètre le polynôme auquel on a retiré un facteur. Au retour des appels récursifs, la fonction reconstruit la liste des facteurs irréductibles.

Dans la fonction `equal_deg_split`, on a besoin de générer des polynômes aléatoirement ce qui nous a conduit à faire une méthode `random` qui crée un polynôme unitaire de degré inférieur à `n` passé en paramètre.

4.5 L'algorithme complet de la factorisation

Pour factoriser un polynôme sur un corps fini, il ne reste plus qu'à combiner ces trois étapes. Dans notre implantation, l'entrée de la méthode `factorise` est un polynôme `a` quelconque, et le résultat est rendu sous forme d'une liste de couple (`poly`,`puiss`) où `poly` est un facteur irréductible de `a` et `puiss` la puissance où il se trouve dans `a`.

```
let factorise(poly in self) in list(self*nat) =
  let rec split(res,liste) =
    match liste with
      | Nil -> res
      | Cons(h,t) -> (
          let poly_sqf = first(h) in
          let deg = scnd(h) in
            split(append((self.split_degree (poly_sqf, deg)),res), t)
      )
  in
  let dest_deg(poly_sqf) =
    split(Nil, self.distinct_degree(poly_sqf))
  in
  let fact(arg) =
    let poly_sqf = first(arg) in
    let puiss = scnd(arg) in
    let aux(x) = crp(x,puiss) in
    let factor = dest_deg(poly_sqf) in
      map(aux, factor)
  in
    let sqf = self.sqrfree(poly) in
      concat(map(fact,sqf));
```

On commence par trouver la décomposition sans facteurs multiples. Puis, on applique la fonction `fact` à tous les polynômes ainsi trouvés. Cette fonction trouve la liste des facteurs irréductibles en appelant la fonction `dest_deg`. Et enfin indique que tous ces facteurs sont à la puissance `puiss` qui avait été trouvée lors de la première étape.

Chapitre 5

Les optimisations

Nous allons commencer par une présentation de l'automorphisme de Frobenius, puis nous verrons sont utilisation pour améliorer les performances de la factorisation.

5.1 Frobenius

Définition 7
L'automorphisme de Frobenius est défini par

$$\sigma : \left\{ \begin{array}{ccc} \mathbb{F}_A & \longrightarrow & \mathbb{F}_A \\ w(x) & \longmapsto & w^q(x) \end{array} \right.$$

où $\mathbb{F}_A = \mathbb{F}_q[x]/\langle a(x)\rangle$ et où $a(x)$ est un polynôme sans facteurs multiples de degré n à coefficients sur $\mathbb{F}_q$.

On peut calculer l'image des puissances de x par σ (car c'est une application linéaire) :

$$\begin{array}{ccccccc} 1 & \overset{\sigma}{\longmapsto} & \sigma_0 & = & 1^q \bmod a(x) & & \\ x & \longmapsto & \sigma_1 & = & x^q \bmod a(x) & & \\ x^2 & \longmapsto & \sigma_2 & = & (x^2)^q \bmod a(x) & = & \sigma_2 \cdot \sigma_1 \bmod a(x) \\ & \vdots & & & & & \\ x^i & \longmapsto & \sigma_i & = & (x^i)^q \bmod a(x) & = & \sigma_{i-1} \cdot \sigma_1 \bmod a(x) \\ & \vdots & & & & & \\ x^{n-1} & \longmapsto & \sigma_{n-1} & = & (x^{n-1})^q \bmod a(x) & = & \sigma_{n-2} \cdot \sigma_1 \bmod a(x) \end{array}$$

Une fois toutes ces valeurs calculées il est simple de calculer $w^q(x) \bmod a(x)$ par un procédé récursif. On peut écrire le polynôme $w(x)$ sous forme de coefficient dominant multiplié par x à la puissance le degré de $w(x)$ plus le reste du polynôme :

$$w(x) = lc(w(x)) \cdot x^{deg(w(x))} + red(w(x))$$

On élève w à la puissance q :

$$w^q(x) \bmod a(x) = \Big((lc(w)) \cdot \Big(x^{deg(w)}\Big) + (red(w))\Big)^q$$

Comme σ est linéaire :

$$w^q(x) \bmod a(x) = (lc(w))^q \cdot \Big(x^{deg(w)}\Big)^q + (red(w))^q$$

donc, grâce au petit théorème de Fermat et aux puissances calculées précédemment, on a :

$$w^q(x) \bmod a(x) = lc(w) \cdot \sigma_{deg(w)} + (red(w))^q$$

On réapplique cet algorithme sur le polynôme diminué de son terme dominant.

Cet algorithme se place naturellement dans l'espèce `polynomes_modulo_unitaire`. Il se décompose en deux parties :
- le calcul des puissances de x par la méthode `puiss_frob`
- l'application de l'automorphisme sur un élément de l'espèce, par la méthode `frobenius`

Le calcul des puissances de x ne doit être réalisé qu'une fois, au moment de la création de la collection.

```
let puiss_frob =
  let deg = degre(themod) in
  let prem = expt_modulo(poly_x,card) in
  let rec calc_tab(prec, i, res) =
    if lt(i,deg) then
      (
       let aux = mult(prem,prec) in
         calc_tab(aux, succ(i), Cons(crp(aux,i),res))
      )
    else
      res
  in
    calc_tab(prem, succ(un), Cons(crp(prem,un),Nil));
```

Les puissances sont stockées sous forme de liste de couple $((x^i)^q,i)$, avec la puissance la plus élevée en tête de liste. Le choix d'utiliser une liste plutôt qu'un tableau se justifie par le fait que cela évite de faire des conversions de degrés en `int`. Et également parce que on accède généralement aux éléments dans l'ordre de la liste. On voit cela dans la méthode `frobenius(w)` qui calcul $w^q \bmod a$.

```
let frobenius(w) =
  let rec trouv(i,liste) =
    match liste with
    | Nil -> foc_error("Non Trouver")
    | Cons(h,t) -> (
        let x = first(h) in
        let d = scnd(h) in
          if eq(d,i) then crp(x,t) else trouv(i, t)
     )
  in
  let rec calc_wq(res, tab, w) =
    let d = degre(w) in
      if is_min(d) then
        plus(res,w)
      else
        let tmp = trouv(d,tab) in
        let x = first(tmp) in
        let stab = scnd(tmp) in
          calc_wq(plus(res, mmult(lc(w), x)), stab, reductum(w))
  in
    calc_wq(zero,self.puiss_frob,w);
```

5.2 Dans l'algorithme de la factorisation en degrés distincts

Nous avons vu que dans l'algorithme de la factorisation en degrés distincts, nous étions amené à calculer w^q mod a. Il est donc naturel d'utiliser Frobenius.

```
let distinct_degree(a) =
  let rec boucle
     (modmon is polynomes_modulo_unitaire(nat*r*self*a),i, a, w, res) =
    let i_fois_2 = dmult(i,deux) in
      if gt(i_fois_2,degre(a)) then
        if is_un(a) then
          match res with
            | Nil -> Cons(crp(a,nat.zero),Nil)
            | _ -> res
        else
          Cons(crp(a,degre(a)),res)
      else
        let nouv_w = modmon.frobenius(w) in
        let ai = gcd(a, moins(nouv_w,poly_x)) in
          if is_un(ai) then
            boucle(modmon, succ(i), a, nouv_w, res)
          else
            let nouv_a = div(a,ai) in
              if is_min(degre(nouv_a))
              then (
                if is_un(nouv_a)
                then Cons(crp(ai,i),res)
                else Cons(crp(a,degre(a)),Cons(crp(ai,i),res))
              ) else (
                let nouv_modmon = polynomes_modulo_unitaire(nat,r,self,nouv_a)
                in
                  boucle(nouv_modmon, succ(i), nouv_a,
                         modulo(nouv_w,nouv_a), Cons(crp(ai,i),res))
              )
  in
    if is_min(degre(a))
    then Cons(crp(a,nat.zero),Nil)
    else
      let modmon = polynomes_modulo_unitaire(nat,r,self,a) in
        boucle(modmon,nat.un, a, poly_x, Nil);
```

On peut remarquer que l'on a du ajouter un paramètre à la fonction `boucle` qui est la collection où on veut calculer Frobenius. Il est important de travailler sur la bonne collection de `polynomes_modulo_unitaire`, c'est pour cela qu'à chaque fois que l'on trouve un nouveau facteur, il faut créer une nouvelle collection. On crée, lorsque c'est nécessaire localement des collections à chaque tour de boucle.

5.3 Dans l'algorithme de la séparation des degrés égaux

Une partie importante du temps de calcul est dédiée au calcul de $x^{\frac{q^d-1}{2}}$. C'est pour cela que l'on va décomposer le calcul de la manière suivante :

$$x^{\frac{q^d-1}{2}} = x^{(\frac{q^d-1}{q-1})(\frac{q-1}{2})}.$$

Nous avons fait apparaître la fraction $\frac{q^d-1}{q-1}$, qui peut se réécrire

$$\frac{q^d-1}{q-1} = 1 + q + q^2 + \cdots + q^{d-1}.$$

Nous pouvons donc écrire

$$\begin{aligned} x^{\frac{q^d-1}{2}} &= x^{(\sum_{i=1}^{d-1} q^i)(\frac{q-1}{2})} \\ &= \left(x \cdot x^q \cdot x^{q^2} \cdots x^{q^{d-1}}\right)^{\frac{q-1}{2}} \end{aligned}$$

Il faut donc calculer $N(x) = \prod_{i=1}^{d-1} x^{q^i}$ qui est la norme. Patrice Naudin et Claude Quitté ont montré dans [?] que l'on peut calculer la trace à la place de la norme, et que de cette façon le calcul est plus efficace.

5.3.1 Utilisation de Frobenius pour le calcul de la trace

Nous avons vu dans le chapitre 4 que la trace est définie par

$$Tr(x) = \sum_{i=0}^{n-1} x^{q^i}.$$

On peut donc l'exprimer en fonction de l'automorphisme de Frobenius.

$$Tr(x) = T_{n-1}(x)$$

où $T_n(x)$ est définie par $\left\{ \begin{array}{lcl} T_0(x) &=& x \\ T_i(x) &=& x + \sigma(T_{i-1}(x)) \end{array} \right.$.

On a donc l'algorithme suivant pour calculer la trace :

```
let trace_mod(modmon is polynomes_modulo_unitaire(nat*r*self*a), u, k) =
  let rec aux(res, cpt) =
    if (nat.lt(cpt,k)) then
      let cour = modmon.frobenius(res) in
        aux(self.plus(u,cour), nat.successeur(cpt))
    else
      res
  in
    aux(u,nat.zero);
```

Une fois cette optimisation réalisée, il reste encore l'élévation à la puissance $\frac{q-1}{2}$ modulo le polynôme $a(x)$ qui coûteuse. Nous allons voir dans le paragraphe suivant comment l'optimiser.

5.4 La multiplication modulo un polynôme unitaire

Nous allons voir ici comment améliorer la multiplication dans l'espèce `polynomes_modulo_unitaire`. Lors de la multiplication de deux polynômes de degrés m et n, le résultat sera de degré $m * n$. Or, nous savons que si nous travaillions modulo un polynôme $a(x)$ de degré d, le résultat sera toujours de degré inférieur à d.

L'optimisation réalisée ici est d'utiliser le fait que les éléments de `polynomes_modulo_unitaire` sont de degré inférieur d, donc si on les multiplie, le degré du résultat est inférieur à $2 * d$. Donc on peut précalculer les puissances de x entre d et $2 * d$, puis lors de la multiplication, remplacer les puissances de x par leurs valeurs.

Donc l'algorithme utilisé est :

```
let multiplie(x,y) =
  let rec trouv(i,liste) =
    match liste with
    | Nil -> foc_error("Non Trouver")
    | Cons(h,t) -> (
        let x = first(h) in
        let d = scnd(h) in
          if eq(d,i) then crp(x,t) else trouv(i, t)
       )
  in
  let rec boucle(max, tab, p, ans) =
    let n = self.degre(p) in
      if geq(n,max)
      then (
        let tmp = trouv(moins(n,max), tab) in
        let x = first(tmp) in
        let stab = scnd(tmp) in
          boucle(max,stab,reductum(p), plus(ans, mmult(lc(p),x)))
      )
      else
        plus(ans,p)
  in
    let p = ann_poly.multiplie(x,y) in
      boucle(degre(themod), self.puiss, p, self.zero);
```

où `self.puiss` est l'appel à la liste des puissances. Comme pour les puissances de Frobenius, on a utilisé une liste de couples (x^i,i) triée par ordre décroissant. Ces puissances sont calculées de la manière suivante :

```
let puiss =
  let prem = ann_poly.oppose(reductum(themod)) in
  let max = degre(themod) in
  let rec boucle(cpt, prec, res) =
    if lt(cpt, max) then
      (
        let cour = mult(poly_x,prec) in
          if eq(degre(cour), max) then
            (
              let aux = plus(reductum(cour), mmult(lc(cour),prem)) in
                boucle(succ(cpt), aux, Cons(crp(aux,cpt),res))
            )
          else
            boucle(succ(cpt), cour, Cons(crp(cour,cpt),res))
      )
    else
      res
  in
    boucle(d.un, prem, Cons(crp(prem,d.zero),Nil));
```

Chapitre 6

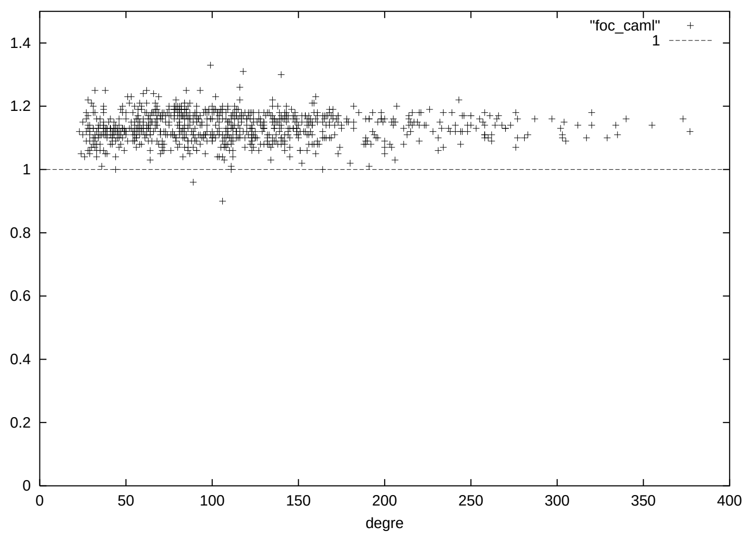
Conclusion

Les algorithmes ayant été à la fois écris en OCaml et en FOC, il est intéressant de comparer leurs performances pour savoir si le code généré par FOC est correct. Il faut aussi pouvoir évaluer les performances avec un autre logiciel de calcul formel ; nous avons choisi Axiom pour cette comparaison. Axiom a été choisi car il utilise une représentation analogue et des algorimes identiques.

6.1 Comparaison FOC par rapport à OCaml

Les deux implantations sont les mêmes, les choix de découpage des structures mathématiques et les algorithmes sont identiques. Donc la comparaison des performances des deux implantations permet de juger si le code généré par FOC est correct.

Dans la courbe suivante, nous comparons le temps de calcul de la factorisation en FOC par rapport au temps en OCaml en fonction du degré des polynômes.

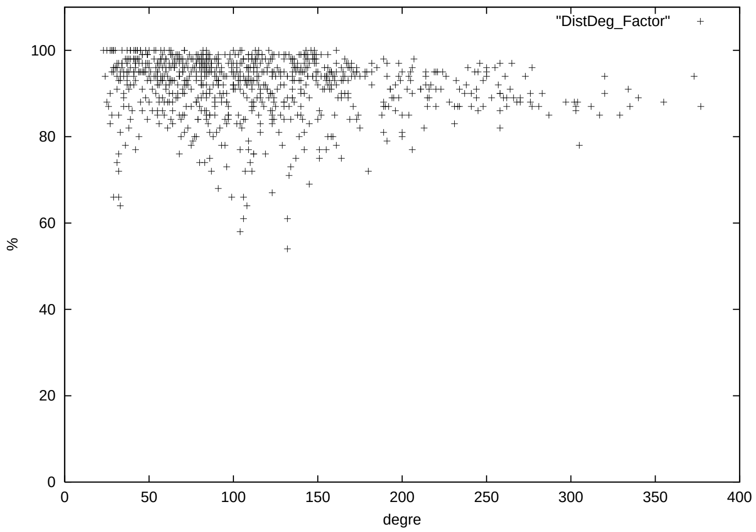


On constate sur cette courbe que les points sont très légèrement au dessus de 1, ce qui signifie que le code FOC est un peu moins rapide que le code écrit directement en OCaml.

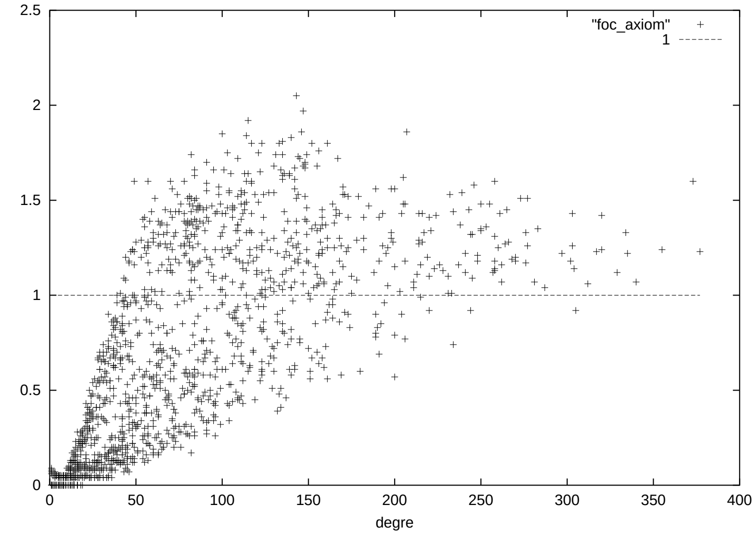
6.2 Comparaison FOC par rapport à Axiom

Dans cette partie nous allons comparer les performances de la factorisation en FOC par à la factorisation en Axiom. D'après l'étude précédente, si on comparait le code OCaml avec Axiom, on aurait des résultats qui serait un peu meilleurs.

Les algorithmes utilisés sont en Axiom semblent être les mêmes que ceux utilisés pour la factorisation en FOC. Le temps passé dans la première étape de la factorisation est négligeable par rapport aux des autres. Nous allons commencer par étudier le cas où la plus grande partie de la factorisation est passée dans la seconde étape. Le graphe suivant représente le pourcentage de temps passé dans la seconde étape par rapport au temps complet de factorisation.



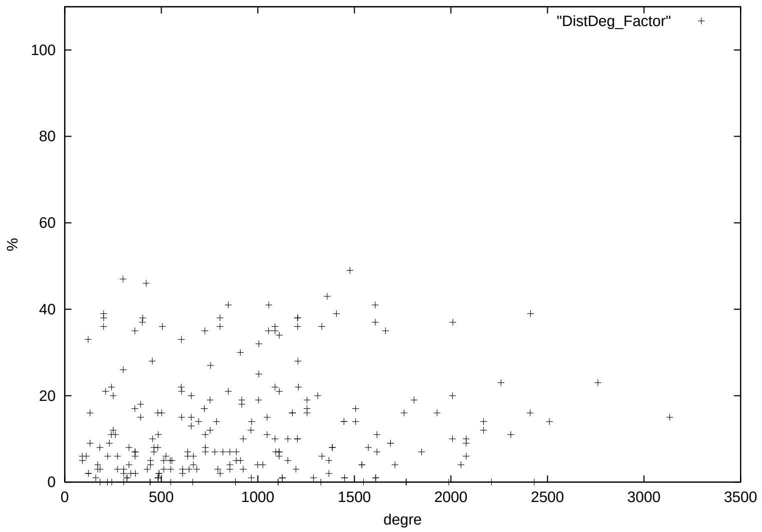
Donc les polynômes que nous étudions ici passe plus de 60% du temps dans la seconde étape. Nous allons comparer pour ces polynômes le temps de la factorisation en FOC par rapport au temps de la factorisation en Axiom.



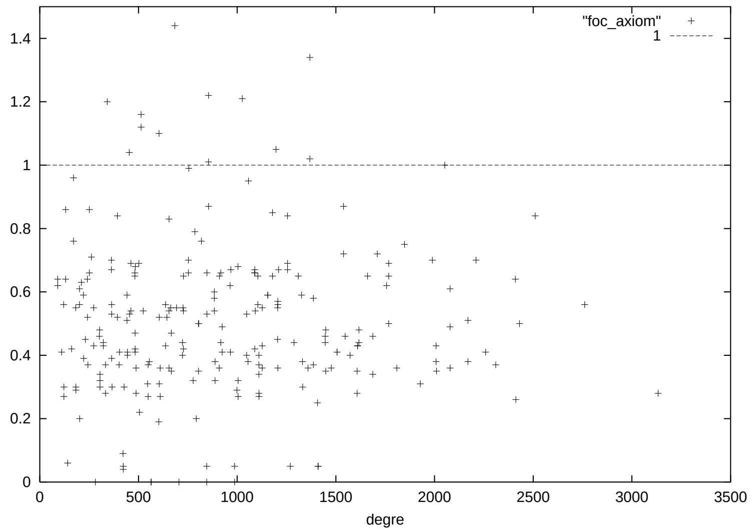
Sur ce graphe, nous voyons que pour des polynômes de faibles degrés, FOC est beaucoup plus rapide que Axiom. Cela est surement dû au fait que Axiom doit précalculer et stocker beaucoup de valeurs.

Puis quand le degré augmente, Axiom devient plus rapide et semble se stabiliser au dessus de 1. Donc, pour la factorisation en degrés distincts, Axiom est plus rapide que FOC.

Nous allons comparer la troisième étape.



Nous voyons que le temps passé dans la seconde étape est inférieur à 50% pour ces polynômes. Comme le temps de la factorisation sans facteurs multiples est n'égligeable par rapport au reste de la factorisation cela signifie que la majorité du temps est passée dans la troisième étape. Donc dans ce cas, le rapport de temps de factorisation entre FOC et Axiom est le suivant.



Nous voyons que FOC est la plupart du temps plus rapide que axiom. Donc globalement les performances de la factorisation sont équivalentes en FOC et en Axiom.

6.3 Bilan

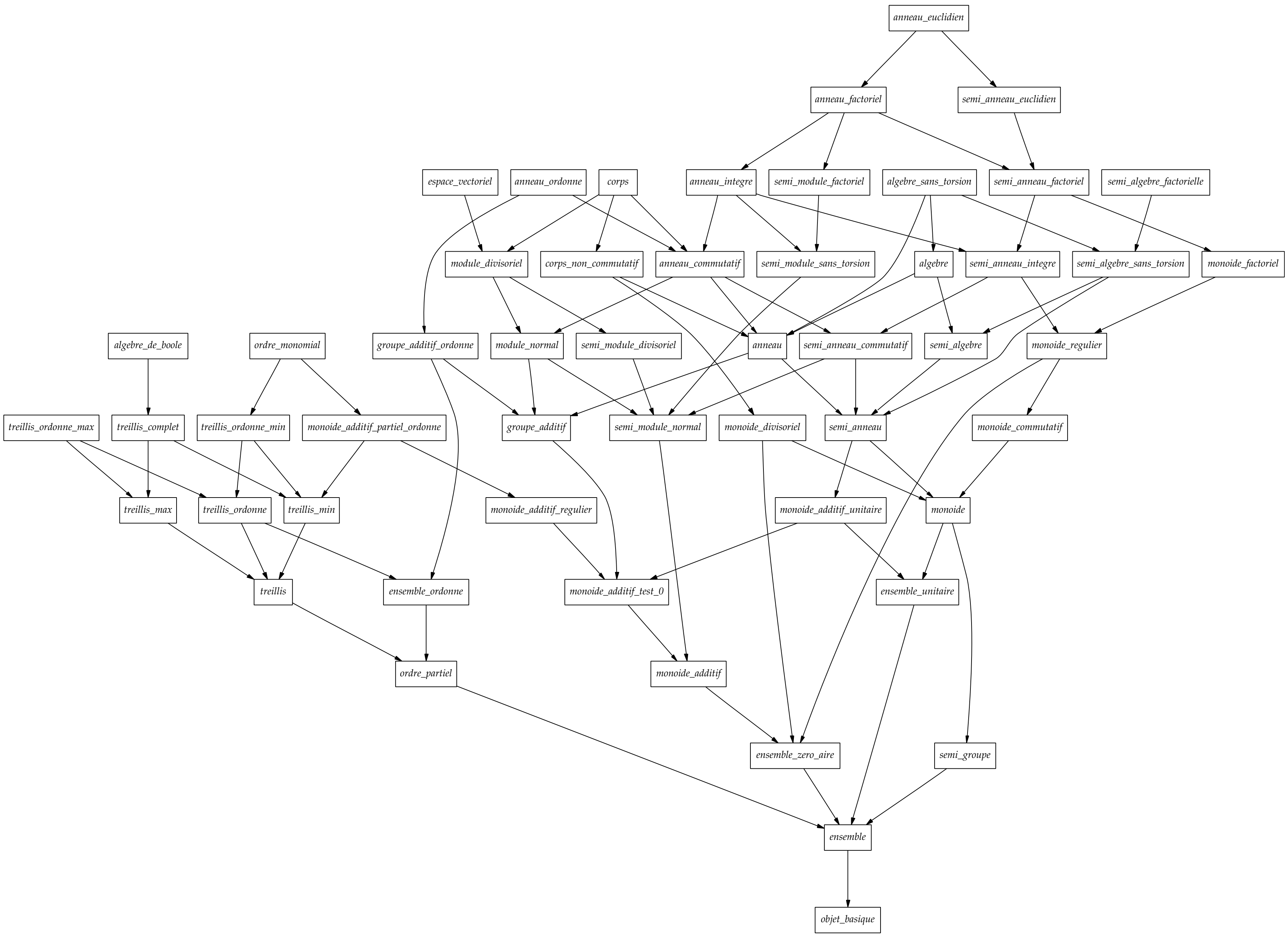
Mon travail au cours de ce stage a permit d'ajouter de nouvelles structures au projet FOC et d'avoir un algorithme efficace pour la factorisation des polynômes à coefficients sur les corps finis. Il a également permi de tester ce nouveau langage et voir s'il était bien adapté pour le calcul formel.

La factorisation represente 963 lignes de OCaml et 914 lignes de FOC. Cela ne représente pas un grosse différence, mais il a été beaucoup plus facil d'écrire le code FOC que le code OCaml.

Sur un plan personnel, ce stage m'a permit de découvrir et d'apprécier le travail dans un laboratoire de recherche.

Annexe A

Structure générale


anneau_euclidien
anneau_factoriel
semi_anneau_euclidien
espace_vectoriel
anneau_ordonne
corps
anneau_integre
semi_module_factoriel
algebre_sans_torsion
semi_anneau_factoriel
semi_algebre_factorielle
module_divisoriel
corps_non_commutatif
anneau_commutatif
semi_module_sans_torsion
algebre
semi_anneau_integre
semi_algebre_sans_torsion
monoide_factoriel
algebre_de_boole
ordre_monomial
groupe_additif_ordonne
module_normal
semi_module_divisoriel
anneau
semi_anneau_commutatif
semi_algebre
monoide_regulier
treillis_ordonne_max
treillis_complet
treillis_ordonne_min
monoide_additif_partiel_ordonne
groupe_additif
semi_module_normal
monoide_divisoriel
semi_anneau
monoide_commutatif
treillis_max
treillis_ordonne
treillis_min
monoide_additif_regulier
monoide_additif_unitaire
monoide
treillis
ensemble_ordonne
monoide_additif_test_0
ensemble_unitaire
ordre_partiel
monoide_additif
ensemble_zero_aire
semi_groupe
ensemble
objet_basique