Projet Calcul Symbolique

Benoit SAINT-ETIENNE
Tom CHAMBARETAUD

Chapitre 1

Polynômes à coefficient dans $\mathbb{F}_2$

1.1 Exercice 1

1.1.1

C'est un ensemble muni de deux opérations binaires rendant possibles les additions, soustractions, multiplications et divisions. Un corps est un anneau commutatif dans lequel l'ensemble des éléments non nuls est un groupe commutatif pour la multiplication.

1.1.2

Pour montrer que $\mathbb{F}_2$ est bien un corps commutatif il faut prouver plusieurs choses :

- Soit le groupe $(\mathbb{F}_2, \oplus)$ on peut écrire : $\forall a \in \mathbb{F}_2,\ a \oplus 0 = a$. Son élément neutre est alors 0.
- Soit maintenant le groupe $(\mathbb{F}_2 \setminus \{0\}, \otimes)$ on peut écrire $\forall a \in \mathbb{F}_2,\ a \otimes 1 = a$. Son élément neutre est alors 1.
- Il faut maintenant prouver la distributivité de $\otimes$.

a	b	c	$b \oplus c$	$a \otimes (b \oplus c)$	$a \otimes b$	$a\otimes$	$a \otimes b \oplus a \otimes b$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	0	1	1	0

1.1.3

L'opération $\oplus$ est assimilable à un *XOR* (un "ou exclusif"), et $\oplus$ est assimilable à un *AND* (un "et").

1.2 Exercice 2

Nous allons utiliser une simple liste d'entiers représentants les degrés du polynôme. Par exemple le polynôme $X^3 + X^2 + X + 1$ sera représenté par une liste croissante $[0; 1; 2; 3]$.

1.2.1

L'interface créée pour satisfaire toutes ces fonctions est la suivante :

```
module type IPoly =
    sig
        type poly

        val poly_empty : poly
        val poly_unit : poly

        val poly_n : int -> poly
        val poly_max_degree : poly -> int
        val poly_first_degree : poly -> int
        val poly_tl : poly -> poly
        val poly_contains : poly -> int -> bool

        val add_poly : poly -> poly -> poly
        val mult_poly : poly -> poly -> poly
        val mult_karatsuba_poly : poly -> poly -> poly
        val newton_poly : poly -> int -> poly
        val quomod_poly : poly -> poly -> poly * poly
        val quomod_newton_poly : poly -> poly -> poly * poly
        val quomod_asc_poly : poly -> poly -> int -> poly * poly
        val euclide_poly : poly -> poly -> poly
        val order_poly : poly -> int

        val irreducible_poly : int -> poly
        val primitive_poly : int -> poly
        val poly_random_not_empty : int -> poly

        val poly_of_string : string -> poly
        val string_of_poly : poly -> string
    end
```

Ainsi

- poly_empty : permet de récupérer un polynôme vide.
- poly_unit : permet de récupérer le polynôme 1.

- poly_n : permet de récupérer le monôme X^n
- poly_max_degree : Renvoie son degré maximal.
- poly_first_degree : Renvoie son degré minimal.
- poly_tl : Renvoie le polynôme privé de son degré minimal.
- poly_contains : Permet de savoir si un polynôme contient un certain degré (si son coefficient n'est pas nul).

— add_poly : effectue la somme de deux polynômes.
— mult_poly : effectue la multiplication naïve de deux polynômes.
— mult_karatsuba_poly : effectue la multiplication par la méthode de Karatsuba de deux polynômes.
— quomod_poly : effectue la division naïve de deux polynômes, renvoie le quotient et le reste.
— quomod_newton_poly : effectue la division rapide par la méthode de Newton de deux polynômes, renvoie le quotient et le reste.
— quomod_asc_poly : effectue la division par la méthode des puissances croissantes de deux polynômes et d'un ordre, renvoie le quotient et le reste.
— euclide_poly : effectue le théorème d'Euclide sur les deux polynômes pour retourner leur PGCD.
— order_poly : renvoie l'ordre du polynôme donné en entrée.

— irreducible_poly : renvoie un polynôme irréductible de dégré n.
— primitive_poly : renvoie un polynôme primitif de dégré n.

— poly_random_not_empty : renvoie un polynôme aléatoire de degré $\leq$ n non nul.

— poly_of_string : permet d'obtenir un polynôme à partir d'une chaîne de caractère. Par exemple le chaîne de caractères "1 + xˆ2" renverra un polynôme stocké dans la mémoire par la liste d'entier : $[0; 2]$
— string_of_poly : permet d'obtenir une chaîne de caractère à partir d'un polynôme. Par exemple le polynôme stocké dans la mémoire par la liste d'entier : $[0; 2]$ renverra la chaîne de caractère "Xˆ2 + Xˆ0"

1.3 Exercice 3

Voici l'algorithme créé pour utiliser la division par les puissances croissantes :

```
let quomod_asc_poly = fun p1 p2 n ->
    let rec aux = fun p1 acc ->
    if ((poly_first_degree p1) > n) then (acc, p1)
    else let q = (poly_first_degree p1) - (poly_first_degree p2)
        in (aux (add_poly (mult_xn_poly q p2) p1) (add_poly (poly_n q) acc))
    in (aux p1 poly_empty)
;;
```

p1 et p2 correspondent aux deux polynômes que l'on veut diviser, et n est l'ordre de la division. L'algorithme s'arrête alors lorsque le reste est divisible par X^{n+1}. Tout d'abord nous effectuons la division entre les deux premières valeurs des polynômes (nous obtenons alors q) ; mais ici comme les coefficients sont dans $\mathbb{F}_2$ il suffit en réalité d'effectuer une simple soustraction de leurs degrés. Nous rappelons ensuite l'algorithme avec comme nouvelle valeur :

$$p1 = p1 - q \times p2$$

$$acc = acc + X^q$$

Chapitre 2

Registre à décalage à rétroaction linéaire

2.1 Exercice 4

2.1.1

La structure de donnée utilisée pour représenter les LFSR sera un enregistrement :

```
type lfsr = {registre : poly; l : int; branchement : poly};;
```

Le registre et les branchements sont alors deux `poly` définies dans le module précédent et `l` représente simplement la longueur de LFSR et est donc représenté par un `int`.

2.1.2

Pour calculer efficacement la nième valeur $r_n (n \geq 0)$ nous avons tout d'abord créé une fonction qui permet de générer la liste de toutes les valeurs de $r_i (i \leq n)$.

```
# rn_list_lfsr lfsr 20;;
- : int list =
[1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0]
```

Et ainsi pour calculer la nième valeur $r_n (n \geqslant 0)$ il suffit alors de récupérer la dernière valeur de la liste. Ce procédé améliore grandement la complexité de la fonction puisqu'elle passe de l'expotionnelle à la linéarité.

```
# rn_lfsr lf1 20;;
- : int = 0
```

Ici `lf1` fait référence au premier LFSR donné en exemple dans le projet.

```
let lf1 = {
    registre = (poly_of_string "1 + x^3 + x^6 + x^9");
    l = 10 ;
    branchement = (poly_of_string "x^1 + x^3 + x^4 + x^7 + x^10")};;
```

2.1.3

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \\ a_l & a_{l-1} & .. & a_3 & a_2 & a_1 \end{pmatrix}, V_n = \begin{pmatrix} r_n \\ r_{n+1} \\ \vdots \\ r_{n+l-3} \\ r_{n+l-2} \\ r_{n+l-1} \end{pmatrix}$$

$$M \times V_n = \begin{pmatrix} r_{n+1} \\ r_{n+2} \\ \dots \\ r_{n+l-2} \\ r_{n+l-1} \\ \alpha_l r_n \times \alpha_{l-1} r_{n+1} \dots \alpha_1 r_{n+l-1} = r_{n+l} \end{pmatrix}$$

Alors nous pouvons dire que : $M \times V_0 = V_1$ et donc : $M^n \times V_0 = V_n$

2.1.4

Les branchements des deux LFSR suivant sont :

```
branchement = (poly_of_string "x^1 + x^3 + x^4 + x^7 + x^10")
branchement = (poly_of_string "x^3")
```

et leurs vingt premières valeurs sont :

```
[1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0]
[1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0; 1; 0; 0]
```

On peut constater que malgré les branchements différents, ces deux LFSR ont les mêmes flux de valeurs : ils ont la même périodicité.

2.2 Exercice 5

2.2.1

$$S(X) \times R(X) = \left(\sum_{i \geq 1} r_i X^i \right) \left(\sum_{i=0}^{l} \alpha_i X^i \right) \tag{2.1}$$

$$= \sum_{i=0}^{l-1} X^i \left(\sum_{j=0}^{i} \alpha_{i-j} r_i \right) + \sum_{i \geq l} X^i \left(\sum_{j=0}^{l} \alpha_i r_{i-j} \right) \tag{2.2}$$

$$= \sum_{i=0}^{l-1} X^i \left(\sum_{j=0}^{i} \alpha_{i-j} r_j \right) + \sum_{i \geq l} X^i \left(r_i + r_i \right) \tag{2.3}$$

$$= \sum_{i=0}^{l-1} \left(\sum_{j=0}^{i} \alpha_{i-j} r_j \right) X^i \tag{2.4}$$

2.2.2

```
val poly_of_lfsr : lfsr -> int * Poly.poly * Poly.poly
```

Cette fonction prend un LFSR et renvoie sa longueur, $G(X)$ et $R(X)$:

```
# let l, g, r = (poly_of_lfsr lf1);;
# string_of_poly g;;
- : string = "X^7 + X^1 + X^0"
# string_of_poly r;;
- : string = "X^10 + X^7 + X^4 + X^3 + X^1 + X^0"
```

2.2.3

```
val lfsr_of_poly : int * Poly.poly * Poly.poly -> lfsr
```

De la même manière à partir de sa longueur, $G(X)$ et $R(X)$ nous pouvons obtenir le LFSR correspondant :

```
# let l = lfsr_of_poly (10, (poly_of_string "X^7 + X^1 + X^0"),
  (poly_of_string "X^10 + X^7 + X^4 + X^3 + X^1 + X^0"))
# string_of_lfsr l;;
- : string =
"LFSR : l = 10; *X^9 + X^6 + X^3 + X^0*; *X^10 + X^7 + X^4 + X^3 + X^1*"
```

2.2.4

Effectivement les deux LFSR produisent le même flux de valeurs et on peut obtenir le triplet $(l, G(X), R(X))$ de second LFSR à partir du premier LFSR, donné dans l'exercice 4, grâce à la division de leurs $G(X)$ et $R(X)$ par leurs $PGCD\,(G(X), R(X)) = T(X)$.

2.2.5

```
val min_lfsr : lfsr -> lfsr
```

Cette fonction prend donc en paramètre un LFSR en renvoie le LFSR minimal qui produit le même flux de valeur :

```
# string_of_lfsr lf1;;
- : string =
"LFSR : l = 10; *X^9 + X^6 + X^3 + X^0*; *X^10 + X^7 + X^4 + X^3 + X^1*"
# string_of_lfsr lf2;;
- : string = "LFSR : l = 3; *X^0*; *X^3*
# string_of_lfsr (min_lfsr lf1);;
- : string = "LFSR : l = 3; *X^0*; *X^3*"
```

Nous voyons donc bien qu'à partir du premier LFSR nous pouvons obtenir le deuxième LFSR.

Chapitre 3

Application à la cryptographie

3.1 Exercice 6

Pour générer un 'bon' LFSR voici la méthode utilisée :

```
let good_lfsr = function n ->
    if (n < 1) then failwith "good_lfsr : n < 1"
    else let r = (primitive_poly n)
    and g = (poly_random_not_empty (n - 1))
    in (lfsr_of_poly (n, g, r))
;;
```

Il suffit de générer un polynôme primitif et un polynôme non nul pour l'état initial. Ensuite à l'aide de ce 'bon' LFSR il est possible de chiffrer des textes. Nous avons pour cela plusieurs fonctions :

- Transformer un nombre en une liste de bits :

  ```
  # bits_of_int 10 7;;
  - : int list = [0; 0; 0; 1; 0; 1; 0]
  ```

  où 7 représente le nombre de bit utilisé dans la représentation du nombre.
- Transformer une liste de bits en un nombre (l'opération inverse) :

  ```
  # int_of_bits [0; 0; 0; 1; 0; 1; 0];;
  - : int = 10
  ```

- Transformer un caractère en une liste de bits :

  ```
  # bits_of_char 'a';;
  - : int list = [1; 1; 0; 0; 0; 0; 1]
  ```

- Transformer une liste de bits en un caractère (l'opération inverse) :

  ```
  # char_of_bits [1; 1; 0; 0; 0; 0; 1];;
  - : char = 'a'
  ```

- Une fonction qui permet d'effectuer un XOR entre les bits de la liste et les r_n correspondant du LFSR :

```
# xor_bits_lfsr glf [1; 1; 0; 0; 0; 0; 1];;
- : int list = [0; 0; 0; 0; 1; 0; 1]
```

Et en effectuant une nouvelle fois l'opération nous obtenons évidemment la même liste :

```
# xor_bits_lfsr glf [0; 0; 0; 0; 1; 0; 1];;
- : int list = [1; 1; 0; 0; 0; 0; 1]
```

Ici glf étant un 'bon' LFSR.

Ainsi, avec ces fonctions, nous avons pu créer les fonctions `encrypt` et `decrypt` qui permettent (respectivement) de chiffrer et déchiffrer une chaîne de caractère. En effet chaque caractère de la chaîne est transformé en une suite de 7 bits extraite du code ASCII qui sont eux même chiffrés grâce au LFSR ; ce qui crée le message chiffré. Il suffit ensuite d'effectuer une nouvelle fois l'opération pour obtenir le message déchiffré.