3.8.4. Séries génératrices

Nous souhaitons représenter des *séries génératrices*. Ce sont des polynômes en une variable z de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n z^n = \alpha_0 + \alpha_1 z + \alpha_2 z^2 + \alpha_3 z^3 + \cdots$$

où les α_i sont, dans le cas qui nous intéresse ici, des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de **coder** de manière compacte des **suites infinies d'entiers**

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \ldots).$$

Exemple

La série génératrice de la suite $(1, 2, 4, 8, 16, \ldots)$ des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n z^n = 1 + 2z + 4z^2 + 8z^3 + 16z^4 + \cdots .$$

Question : comment représenter des séries génératrices ?

Il est impossible de les représenter par des listes à cause du caractère **infini** de ces objets.

Réponse : par une **fonction** qui à tout entier positif n associe le coefficient α_n de z^n.

Ceci est réalisé par le type

```
type series_generatrices = int -> int
```

En effet, pour connaître une série génératrice, il faut et il suffit de connaître pour chaque n le coefficient α_n du monôme $\alpha_n z^n$.

Voici une fonction renvoyant la liste des `n` premiers coefficients d'une série génératrice `sg` :

```
let premiers_coefficients n sg =
    List.init n sg
```

Exemple

La série génératrice des puissances de 2 est ainsi codée par

```
# let puissances_2 =
    fun n -> 1 lsl n;;
val puissances_2 : int -> int = <fun>
```

Nous avons

```
# puissances_2 |> premiers_coefficients 10;;
- : int list = [1; 2; 4; 8; 16; 32; 64; 128; 256; 512]
```

Il existe plusieurs **opérations** sur les séries génératrices :

1. *addition* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)+\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)z^k \ ;$$

2. *produit d'Hadamard* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)\boxtimes\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k z^k \ ;$$

3. *multiplication* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}z^k.$$

Il est possible de les implanter simplement en utilisant des fonctions d'ordre supérieur.

Voici trois manières équivalentes de définir une fonction `addition` qui renvoie la série génératrice résultat de l'addition entre les séries génératrices `s1` et `s2` :

```
let addition s1 s2 =
    fun k -> (s1 k) + (s2 k)
```

```
let addition s1 s2 n =
    (s1 n) + (s2 n)
```

```
let addition s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res
```

Exemple

```
# addition puissances_2 puissances_2 |> premiers_coefficients 10;;
- : int list = [2; 4; 8; 16; 32; 64; 128; 256; 512; 1024]
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =
    fun k -> (s1 k) * (s2 k)
```

Exemple

```
# produit_hadamard puissances_2 puissances_2 |> premiers_coefficients 10;;
- : int list = [1; 4; 16; 64; 256; 1024; 4096; 16384; 65536; 262144]
```

L'implantation de la multiplication est un peu plus technique dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let multiplication s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then 0
            else (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        aux 0
    in
    resultat
```

Exemples

```
# let sg_un = fun _ -> 1;;
val sg_un : 'a -> int = <fun>
# sg_un |> premiers_coefficients 10;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]

# let sg_un_2 = multiplication sg_un sg_un;;
val sg_un_2 : int -> int = <fun>
# sg_un_2 |> premiers_coefficients 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

3.9. Polymorphisme

3.9.1. Notions de base

Une entité est dite *polymorphe* si elle ou certains de ses composants sont d'un type non fixé.

1. Une *fonction polymorphe* est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

Exemple

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

2. Un *type polymorphe* est un type paramétré.

Exemple

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

3. Une *valeur polymorphe* est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé.

Exemple

```
# Vide;;
- : 'a liste = Vide
```

En OCaml, le polymorphisme est *paramétrique* : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« *tout ou un* »).

De cette manière, pour **déterminer le type d'un paramètre** `x` d'une fonction correctement typée, le système de typage fonctionne (de manière très simplifiée) ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de `x`), alors `x` est du type le plus général `'a`.

Plusieurs variables de types `'a`, `'b`, `'c`, *etc.*, peuvent coexister dans l'expression de type d'un objet polymorphe.

Beaucoup de fonctions de la bibliothèque standard sont polymorphes. Parmi elles :

- `(=) : 'a -> 'a -> bool` qui teste l'égalité entre deux valeurs d'un même type.
- `(<>) : 'a -> 'a -> bool` qui teste la différence entre deux valeurs d'un même type.
- `compare : 'a -> 'a -> int` qui est telle que `compare x y` renvoie `-1` (resp. `1`) si `x` est strictement inférieure (resp. supérieure) à `y` et `0` sinon.

  Ceci est une **fonction générique de comparaison** entre deux valeurs d'un même type.
- `fst : 'a * 'b -> 'a` qui renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
- `snd : 'a * 'b -> 'b` qui renvoie la 2e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
- `(@@) : ('a -> 'b) -> 'a -> 'b` qui est telle que `f @@ x` renvoie `f x`.
- `(|>) : 'a -> ('a -> 'b) -> 'b` qui est telle que `x |> f` renvoie `f x`.
- `ignore : 'a -> unit` qui est telle que `ignore x` renvoie `()`.

3.9.2. Monoïdes et exponentiation dichotomique

Lorsque x est un nombre et n est un entier positif, le calcul de x^n se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en la fonction

```
let rec puissance x n =
    if n = 0 then
        1
    else
        let tmp = puissance x (n / 2) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

L'opération d'exponentiation peut s'appliquer à d'autres objets que des nombres, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

En d'autres termes, ces objets doivent former une structure de monoïde.

Nous souhaitons ainsi écrire une fonction polymorphe `puissance_polymorphe` de type

`('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e`

où

- le 1er paramètre de type `'e -> 'e -> 'e` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;
- le type de retour est `'e`.

La valeur renvoyée est x^n.

Nous obtenons ainsi la fonction polymorphe

```
let rec puissance_polymorphe op unite x n =
    if n = 0 then
        unite
    else
        let tmp = puissance_polymorphe op unite x (n / 2) in
        if n mod 2 = 0 then
            op tmp tmp
        else
            op (op tmp tmp) x
```

Exemples

```
# puissance_polymorphe (+) 0 1 6;;
- : int = 6

# puissance_polymorphe ( * ) 1 2 10;;
- : int = 1024
```

```
# puissance_polymorphe (^) "" "abb" 4;;
- : string = "abbabbabbabb"

# puissance_polymorphe (||) false false 293898273;;
- : bool = false
```

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Pour cela, on représente les monoïdes au moyen du **type enregistrement**

```
type 'a monoide = {
    op : 'a -> 'a -> 'a;
    unite : 'a
}
```

La fonction précédente devient

```
let rec puissance_polymorphe monoide x n =
    if n = 0 then
        monoide.unite
    else
        let tmp = puissance_polymorphe monoide x (n / 2) in
        if n mod 2 = 0 then
            monoide.op tmp tmp
        else
            monoide.op (monoide.op tmp tmp) x
```

3.9.3. Listes génériques

Soit le type des **listes génériques**

```
type 'e liste =
    |Vide
    |Cellule of 'e * 'e liste
```

Nous souhaitons écrire une fonction polymorphe `appartient_liste` de type

```
'e -> 'e liste -> bool
```

telle que `appartient_liste x lst` teste la présence de l'élément `x` dans la liste `lst`.

```
let rec appartient_liste x lst =
    match lst with
        |Vide -> false
        |Cellule (y, _) when y = x -> true
        |Cellule (_, reste) -> appartient_liste x reste
```

Ceci se base sur le fait que le test d'égalité `=` est **polymorphe** : il est de type `'a -> 'a -> 'a`.

De plus, comme ce test est **présent de manière générique**, il n'est donc pas nécessaire d'avoir un paramètre supplémentaire pour `appartient_liste` destiné à recevoir une fonction de test d'égalité.

Nous souhaitons maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le **plus grand élément d'une liste générique**. Cette fonction est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e option
```

où

- le 1er paramètre de type `'e -> 'e -> 'e` contient une fonction qui prend comme arguments deux éléments et renvoie le plus grand. Il code l'**opération « max »** d'une relation d'ordre totale ;
- le 2e paramètre de type `'e liste` contient la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est une **valeur optionnelle** de type `'e`. La valeur encapsulée est le plus élément de la liste si la liste est non vide et est `None` sinon.

```
let maximum_liste f_max lst =
    let rec aux lst max_prefixe =
        match lst with
            |Vide -> max_prefixe
            |Cellule (x, reste) -> aux reste (f_max max_prefixe x)
    in
    match lst with
        |Vide -> None
        |Cellule (x, reste) -> Some (aux reste x)
```