TME5 - Évaluation non-stricte et structures infinies

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TME nous allons construire des outils permettant de manipuler de calculer avec des séries formelles et des polynômes.

Démarrage du projet

Le projet se nomme simplement `Formel` et se décompose en deux parties :

— une première partie permettant d'effectuer des calculs symboliques sur des *séries formelles*
— une seconde partie permettant des calculs similaires sur des *polynômes* implémentés de manière *creuse*

En guise de préliminaire, il faudra créer un projet *Stack* (en choisissant toujours le *resolver* `lts-17.2`) avec deux modules sources (dans `src/`):

— le module `GFun` pour les séries génératrices
— le module `Poly` pour les polynômes

On développera aussi des tests avec `HSpec` en s'inspirant du TME 3.

Les bibliothèques tierces utilisables dans ce projet sont :

— `containers` pour les séquences, *maps*, etc.
— `hspec` pour les tests unitaires et BDD

Il ne faut évidemment pas utiliser de bibliothèques de calcul formel.

Module GFun

Une *série formelle* est une généralisation des polynômes de la forme :

$$\sum_{n}^{+\infty} c_n z^n$$

où c_n est suite à valeur dans un anneau commutatif R, z est une variable symbolique et n parcourt les entiers naturels. Dans l'implémentation, on considérera que les c_n ont pour type une instance de la classe de types `Num`.

En Haskell, on peut exploiter le mode d'évaluation non-stricte pour implémenter les séries génératrices. Pour notre projet nous utiliserons le type suivant:

```haskell
data Serie a = Z a (Serie a)
```

On peut interpréter ce type comme une occurrence de `Z` avec un coefficient de type `a`, suivi du reste de la série.

Remarque : On ne demande pas au type `a` d'instancier des classes de types (entre autres, `Num`) lors de la définition du type. Contraindre des types sans raison "immédiate" (ex.: le besoin d'utiliser une fonction particulière) est considéré comme une mauvaise pratique en Haskell. Les contraintes doivent être retardées jusqu'au moment où elles sont nécessaires: à la définition de fonctions ou d'instanciation.

Un premier exemple de série

On peut construire la série où tous les coefficients sont à 1 de la façon suivante :

```
uns :: Serie Integer
uns = Z 1 uns
```

pour pouvoir observer les séries, on définit une fonction `prend` telle que:

```
>>> prend 10 uns
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Il s'agit de la série *génératrice* (la série formmelle associée à une suite) de la suite constante de valeur 1 :

$1.z^0 + 1.z^1 + 1.z^2 + \cdots + 1.z^n + \cdots$

Pour l'affichage des séries, on ne retiendra que les 10 premiers termes...

Par exemple :

```
>>> show uns
"1.z^0 + 1.z^1 + 1.z^2 + 1.z^3 + 1.z^4 + 1.z^5 + 1.z^6 + 1.z^7 + 1.z^8 + 1.z^9 + ..."
```

On souhaite également définir une fonction de *map* pour appliquer une même fonction unaire à tous les éléments d'une série :

```
gmap :: (a -> b) -> Serie a -> Serie b
```

Par exemple :

```
>>> show (gmap (*2) ones)
"2.z^0 + 2.z^1 + 2.z^2 + 2.z^3 + 2.z^4 + 2.z^5  + 2.z^6 + 2.z^7 + 2.z^8  + 2.z^9 + ..."
```

Remarque : nous verrons au prochain cours la typeclasse `Functor` qui permet d'utiliser la fonction générique `fmap`.

Opérations sur les séries

On désire instancier la class `Num` avec les séries dont les coefficients supportent également cette classe.

Les opérations se font membre à membre, à l'exception de la multiplication de deux séries, qui exploite le calcul semi-formel suivant :

```
`(Z c1 s1) * (Z c2 s2) = Z (c1 * c2) (gmap (*c1) s2 + gmap (*c2) s1 + Z 0 (s1 * s2))`
```

Pour les opérations "bizarres" (`signum` par exemple), l'implémentation est libre.

On ajoutera la *dérivation formelle* d'une série formelle, avec la fonction de signature suivante :

```
derive :: Num a => Serie a -> Serie a
```

Résumé

- Instancier `Show`, `Num` et `Functor` avec `Serie`.
- Proposer une batterie de tests *HSpec* pour les opérations de ces classes de types.

** Remarque ** : Les tests sur les `Serie` demandent à ce type d'instancier `Eq`. L'égalité dérivée d'Haskell est stricte, il faut donc instancier à la main cette classe de types (par exemple en comparant les 100 premiers termes des deux séries).

Module Poly

L'objectif de ce module est de fournir une implémentation **creuse** des polynômes, c'est à dire d'expressions de la forme :

$$c_n X^n + c_{n-1} X^{n-1} + \cdots + c_2 X^2 + c_1 X^1 + c_0$$

où :

X est une unique *variable symbolique* - les $c_0, \ldots, c_n$ sont des nombres appelés *coefficients* - le plus grand j tel que $c_j \neq 0$ est appelé le *degré* du polynôme

Voici un exemple de polynôme de degré 4 :

$$X^4 + 2X^2 + 4$$

On identifie $c_n x^n + \cdots + c_{k+1}x^{k+1} + 0x^k + c_{k-1}x^{k-1} + \cdots + c_0$

avec $c_n x^n + \cdots + c_{k+1}x^{k+1} + c_{k-1}x^{k-1} + \cdots + c_0$ (ce sont les mêmes polyômes).

Le *polynôme nul* est le polynôme dont tous les coefficients sont nuls.

Un polynôme est *canonique* quand aucun de ses coefficents n'est nul.

Concernant la représentation en Haskell, il existe plusieurs possibilités mais nous allons choisir une représentation creuse à partir du type

```
data Poly a = Nul | Poly (Int, Map Int a)
```

où `a` est le type des coefficients.

- `Nul` est l'état remarquable du polynôme nul,
- `Poly (d, m)` est un polynôme de degré au plus `d` et la table `m` associe à des entiers les coefficients correspondants.

A noter qu'un polynôme n'est pas forcément canonique: `m` peut associer des coefficients nuls à certaines valeurs. Deplus, on veut que les clefs de `m` ne soient pas plus grandes que `d` (c'est-à-dire que le polynôme soit bien de degré *au plus* `d`).

Proposer un invariant des polynomes qui réifie cette propriété.

Opérations arithmétiques

Le type `Poly` doit instancier la classe `Num`. Comme pour les séries, les opérations "bizarres" sont laissées libres.

Pour être efficace, il est nécessaire de maintenir la structure creuse des polynômes au cours des calculs, ainsi les opérations arithmétiques doivent renvoyer des polynômes *canoniques*. Il est utile de définir une fonction `canonise` qui met un polynôme sous forme canonique.

Pour certaines opérations, il peut être utile d'implémenter une fonction de *map* sur `Poly` (que l'on appellera par exemple `pmap`).

Dérivation formelle

Définir la *dérivée* formelle (symbolique) d'un polynôme (fonction `derive`).

Ajouter une post-condition à `derive` qui assure que $(P.Q)' = P'.Q + P.Q'$

Syntaxe concrète

On veut pouvoir manipuler des fichiers et des entrées/sorties avec polynômes. Ainsi, le type `Poly` doit instancier `Show` et `Read`.

On suppose que le format de représentation des polynômes est

```
"<cn>.X**<an> + <cn-1>.X**<an-1}> + ... + <c1>.X**<a1> + <c0>"
```

- les coefficients `c_i` sont représentés par `show` de leur valeur et sont séparés de la variable symbolique par un `.`
- `<ci>.X**<ai>` est représenté par `1.X**<ai>` si `<ci>` vaut `1` (et pas par `X**<ai>`)
- les puissances sont séparées de la variable symbolique par `**`
- les monômes sont séparés les uns des autres par des `+`
- les degrés `<ai>` sont tous différents et rangés par ordre décroissant
- `<c0>` est toujours présent, même s'il est nul
- le polynome nul est représenté par `()`

Donner l'instanciation de `Show`.

Pour `Read`, une analyse syntaxique complète des chaînes d'entrée est laissée en **défi**. On demande, pour un rendu standard, de pouvoir lire correctement les polynômes dont les coefficients sont des entiers (positifs ou négatifs) écrits dans le format donné plus haut.

Résumé

— Instancier `Num` avec `Poly`.
— Proposer une batterie de tests *HSpec*.
— Instancier `Show` avec `Poly`.
— Instancier `Read` avec `Poly`, en se restreignant à des polynomes entiers, sans analyse syntaxique complexe.
— Proposer un complément de tests *HSpec*.