Projet : Résolution d'un système linéaire et vérification automatique

Nathanael Bayard – L2 Info -- 2017/2018 -- Méthodes Matricielles

Python

J'ai choisi la version 2.7 de Python pour écrire la partie 2, afin de garder une compatibilité optimale avec les parties 1 et 3, écrites dans *Sage*.

Linux

Le programme a été écrit sur *Xubuntu 16.04*, et testé avec succès sur l'un des postes d'une des salles de TP pour la partie Python (lancé depuis le terminal avec la commande `python2.7 Part2.py`). Les parties sur Sage sont supposées indépendantes du système qui les lance.

Notations

Dans ce PDF et dans le code, les notations suivantes relatives au typage seront utilisées :

- Le symbole `:` sera utilisé pour définir le type d'une valeur. On notera `x : t` pour exprimer que `x` est du type `t`. Par exemple : `"hello" : String`.
- La signature du type d'une fonction `f(x, y, z)` sera notée ainsi :

  ```
  f : X . Y . Z -> T
  ```

  où `X` / `Y` / `Z` est le type du paramètre `x` / `y` / `z`, et `T` est le type de l'*output* de `f`.
- Si une fonction produit des effets de bords (*side effects*), on rajoutera `[IO]` (pour *input/output*) à la fin de la signature:

  ```
  f : X . Y . Z . -> T [IO]
  ```
- Le type de retour d'une fonction qui ne renvoie rien est `Void`
- Par convention, dans les signatures de type, tout identificateur commençant par une minuscule est une variable de type (*c.f.* plus bas concernant la polymorphie). Le type peut varier sans que la fonction s'en mêle. Par contraste, les identificateurs qui commencent par une majuscule seront des types monomorphiques ou bien des contraintes sur les variables de type (*c.f.* plus bas).
- En Python, une fonction peut très bien avoir des paramètres étant eux-même des fonctions (*cf* **Style de Programmation**). Par exemple voici le type de la célèbre fonction `map` :

  ```
  map : (a -> b) . List a -> List b
  ```

  Son premier paramètre est une fonction qui prend un object d'un type indéterminé `a` et renvoie une valeur de type `b`.
- On peut remarquer que le type de `map` est polymorphique de manière générique : les variables `a` et `b` peuvent représenter n'importe quel type sans que `map` ait besoin d'être modifiée au niveau de son algorithme interne. L'alignement des types (c'est-à-dire le fait que la variable de type `a` dans le type du premier paramètre soit le même que la variable `a` dans le type du second paramètre, et *idem* pour `b`) est tout ce qui compte pour que cela fonctionne.

  Certaines fonctions sont cependant polymorphiques *ad-hoc*: certaines variables de types sont sujettes à des conditions, comme la condition d'être d'un type qui représente un nombre, que ce soit `Integer`, `Float`, `Fractional` ou autre. On notera cette polymorphie avec des crochets, de cette manière :

  ```
  add : [Num a] a . a -> a
  ```

  La fonction `add` prend deux paramètres de même type `a` quelconque mais qui doit tout de même être un type de **nombre**, et qui retournera une nouvelle valeur du même type `a`. `Num` représente une **classe de type** ou encore une **contrainte de type**

au sens de *Haskell*, c'est-à-dire un ensemble de types restreint pour lesquelles la fonction `add` fonctionne, et ce de manière polymorphique, ou dit autrement une contrainte sur la polymorphie de la variable de type `a` dans le type de `add`.

`Num` sera la seule classe de type utilisée dans le code.

- Quelques types composites usuels et leur notation :

```
x : a
y : a
z : b
[x, y] : List a
(x, y, z) : (a, b, b)
3 : [Num n] n
"abc" : String
True : Bool
```

Le type des nombres entiers sera `Int`.

- Quelques *aliases* qui seront également utilisés et qui permettent de différencier par exemple entre la nature d'une liste de nombre et la nature d'un vecteur matriciel, même s'ils ont la même valeur (et le même type) pour Python.

  La notion d'alias implique que ce n'est pas une erreur de type, en particulier, d'utiliser `List (List n)` et `Matrix n` de manière interchangeable.

```
Matrix n = List (List n)
Vector n = List n
Family n = List (Vector n)
RawSystem n = (Matrix n, Vector n)
# ^-- une matrice du côté gauche et un vecteur du côté droit
#     (ce ne sont que des listes, pas des objets de Sage, d'où le "raw").
RawSolution n = Maybe (Family n, Vector n)
# ^-- la famille est celle des vecteurs de la base du noyau trouvée
#     le vecteur est la solution particulière trouvée
Index = Int
Dim = Int
```

  `Index` représente un entier supérieur ou égal à zéro, et `Dim` (la dimension d'un vecteur, d'une matrice, etc) représente un entier strictement supérieur à zéro.

  Je mentionnerai aussi `Row n` et `Column n`, tous deux synonymes pour `List n` et qui représentent respectivement une ligne ou colonne extraite d'une matrice. Du fait que les matrices sont enregistrées ligne par ligne, on pourrait ainsi aussi écrire :

```
Matrix n = List (Row n)
```

- Du fait que Python n'offre pas de mécanisme pour créer une fonction partielle de manière automatique, plusieurs fonctions ont dues être écrites de manière variadique: elles prennent typiquement en paramètres:
  - une autre fonction, appelons-la `f`
  - le premier argument de `f`,
  - et ensuite elle prennent un nombre indéterminé de paramètres qui vont eux aussi être passés à `f`.

  Par souci de simplicité, j'ai omis l'aspect variadique de ces fonctions dans leurs signatures de type. Elles seront marqués comme prenant juste deux paramètres : une fonction d'un argument, et l'argument qui sera passé à cette fonction.

- Suivant la tradition de *Haskell*, parfois une valeur d'un type `List t` sera nommé `xs` ou `ys`, dans l'idée que cela veux dire "plusieurs x/y". C'est bien sûr surtout dans le cas où on ne sait pas ce que contient la liste, parce que la fonction qui contient la variable est polymorphique.

- Certaines fonctions peuvent, dans des conditions tragiques, lever une exception. J'ai décidé de ne pas le noter dans les signatures de mes fonctions, pour la raison suivante:

  Dans mon code, aucune exception ne participe au fonctionnement normal du programme: si une exception est levée, c'est que le code ne fonctionne pas correctement. Par contraste, une fonction voulant renvoyer l'idée qu'elle ne peut accomplir son travail, mais d'une manière prévisible et normale, renverra plutôt une valeur de type `Maybe` ou `Either`. (*c.f.* plus bas.)

  L'exemple évident ici est une fonction qui résoud un système d'équation, et veut renvoyer l'idée que l'ensemble des solutions est l'ensemble vide: il semblerait absurde de lever une exception dans ce cas-là. (c'est d'ailleurs pourtant ce que fait Sage…)

Style de programmation

J'ai choisi d'adopter un style **purement fonctionnel** autant que possible, car je pense que cela permet de raisonner sur le code de manière bien plus efficace.

Le paradigme **fonctionnel** permet aussi et surtout d'atteindre des niveaux d'abstractions plus élevés, qui permettent une réutilisation optimale du code ainsi qu'une délégation de tâches répétitives et prones aux erreurs, comme la gestion des indices dans des boucles `for`.

Le paradigme **purement** fonctionnel implique que toute variable est immutable et sa valeur ne change pas tout au long de la durée de vie de la variable. Cela a pour conséquence de ne jamais risquer qu'une fonction modifie la valeur d'un objet donné en paramètre d'une manière qu'il serait difficile de suivre ou vérifier (fléau notoire de la programmation objet).

Toute fonction pure étant complètement déterminée par le lien entre *input* et *output*, les test unitaires en sont d'autant plus faciles à réaliser car aucun environnement extérieur n'a besoin d'être contrôlé ou simulé: si la fonction et pure et fait ce qu'il faut avec tous les paramètres possibles donnés en entrée, on peut alors être catégoriquement sûr qu'elle fera ce qu'il faut partout, dans n'importe quel programme, quel que soit les valeurs d'autres variables dans son environnement extérieur.

Les outils les plus classiques des langages fonctionnels se révèlent naturellement utiles dans mon programme : entre autres, `map`, `filter`, `reduce` (parfois aussi appelé `fold`) sont utilisés abondament dans le programme. Beaucoup de fonctions prennent d'autres fonctions en paramètres (en tant que valeurs de première classe), et/ou retournent une fonction en valeur de sortie.

En Python, écrire de manière purement fonctionnelle n'est pas chose aisée. Aussi beaucoup de fonctions admettent la modification de variables pourvues que ces modifications ne se déroulent que lors de la **construction** de la valeur. Par exemple, une implémentation de `map` pourrait ressembler à :

```
def map(f, xs):
    out = []
    for x in xs:
        out.append(f(x))
    return out
```

La variable `out` est mutée tout au long de la fonction, mais on remarquera que cela correspond bien à la phase de **construction** de la valeur de sortie, et qu'aucune valeur donnée en paramètres n'est modifiée. Dans ce genre de cas, on omettra d'écrire `[IO]` à la fin de la signature de la fonction.

`Maybe` et `Either`

J'ai choisi d'utiliser deux types algébriques très utilisés dans les langages dits purement fonctionnels comme *Haskell* (c'est-à-dire, qui évitent autant que possible tout effet de bord ou *side effects*), qui sont les types `Maybe` et `Either`, qui permettent de gérer élégamment et automatiquement le séquençage d'opérations qui chacunes peuvent échouer et donc court-circuiter la série complète d'opération, et renvoyer une valeur qui représente une erreur (dans le cas de `Maybe`), ou une valeur représentant un message d'erreur correspondant à la première erreur rencontrée (dans le cas de `Either`). Ces types remplissent des rôles équivalents aux *exceptions* et aux valeurs arbitraires renvoyées lorsqu'une fonction ne peut faire son travail, comme lorsqu'une recherche renvoie `-1` ou `null` en cas d'échec.

Maybe

`Maybe` est en réalité un *constructeur de type*, soit une fonction dit *type-level* qui prend un type `t` en paramètre et retourne un nouveau type `Maybe t` en sortie. Une valeur `x : Maybe t` **représentera** ou bien une valeur de type `t`, ou bien une "erreur" (dans un sens large qui dépendra de l'utilisation).

Les valeurs possibles du type `Maybe t` seront ou bien de la forme `Just(y)` avec `y : t`, ou bien seront la valeur unique et arbitraire dénotée `Nothing`, qui représente la notion d'erreur. `Maybe t` est un type qui ne fait donc qu'envelopper (*wrapping*) de manière transparente une valeur d'un type `t` quelconque, en ajoutant la possibilité qu'au lieu d'une valeur `Just(y) : Maybe t`, on ait la valeur `Nothing : Maybe t`.

Tout l'intérêt de `Maybe` réside dans deux fonctions fondammentales que j'ai nommé `maybeApply` et `maybeDo`. Ces fonctions ont été écrites en Python sous la forme de méthodes de la classe `Maybe` principalement pour des raisons de praticité syntaxique.

Signature de `maybeApply` (appelée `fmap` en *Haskell*):

```
`maybeApply : Maybe a . (a -> b) -> Maybe b`
```

L'algorithme interne est trivial et consiste à ne rien faire si la valeur du premier paramètre est `Nothing`. Axiomatiquement :

```
maybeApply(Nothing, f) = Nothing
maybeApply(Just(y), f) = Just(f(y))
```

Signature de `maybeDo` (appelée `bind` ou `>>=` en *Haskell*):

```
`maybeDo : Maybe a . (a -> Maybe b) -> Maybe b`
```

La principale différence entre `maybeDo` et `maybeApply` tient au fait que le second paramètre de `maybeDo` est une fonction qui peut renvoyer `Nothing`. Algorithme interne de `maybeDo` :

```
maybeDo(Nothing, f) = Nothing
maybeDo(Just(y), f) = f(y)
```

Either

`Either` est très similaire à un `Maybe` pour lequel la valeur `Nothing` pourrait contenir une valeur, qui servirait de message d'erreur. `Either` prend deux types en paramètres, le type du message d'erreur attendu et le type de la valeur contenue si tout se passe bien (dans cet ordre). On parlera par exemple de `Either String Int`, `String` étant le type du message d'erreur, et `Int` le type de la valeur contenue s'il n'y a pas d'erreur.

Pour le type `Either e a`, les **constructeurs** de valeurs sont respectivement `Left(x)` avec `x : e` et `Right(y)` avec `y : a`. (La variable `e` signifie **e**rreur)

Comme pour `Maybe`, ce type prend toute son utilité lorsqu'on lui associe deux fonctions que j'ai appelé `eitherApply` et `eitherDo`. Comme dans le cas de `maybeApply/Do`, toute valeur de la forme `Left(x) : Either e a` ressortira inchangée au niveau de sa valeur (bien que son type change depuis `Either e a` vers `Either e b`):

```
eitherApply : Either e a . (a -> b) -> Either e b
eitherApply(Left(x), f) = Left(x)
eitherApply(Right(y), f) = Right(f(y))

eitherDo : Either e a . (a -> Either e b) -> Either e b
eitherDo(Left(x), f) = Left(x)
eitherDo(Right(y), f) = f(y)
```

La classe `System`

J'ai utilisé cette classe pour bien séparer dans l'algorithme de la méthode de Gauss, les lignes sur lesquelles un pivot avait déjà été trouvé, et les lignes restantes. Cela permet de ne rechercher le prochain pivot que dans les lignes qui ne sont pas **déjà** "pivotantes". A la fin du processus d'échelonisation, et si tout s'est bien passé (pas d'équation `0 = n`, `n != 0` rencontré), les éventuelles lignes "non pivotantes" restantes sont forcément pleines de zéroes, et peuvent être ignorées durant les phases suivantes (normalisation et extraction de solution).

Au début de l'algorithme, j'ai choisi de fusionner la matrice de gauche avec le vecteur de droite, soit `A` et `Y` dans `AX = Y`, afin que toute opération sur une ligne de la matrice résultante soit effectuée autant sur `A` que sur `Y`. Evidemment cela a nécessité d'éviter que l'algorithme ne recherche des pivots dans la colonne correspondant au vecteur de droite, mais c'est un moindre mal.

Commentaires concernant la partie 3

Format de sortie pour les solutions trouvées dans la partie 2

Le format choisi pour enregistrer les systèmes et leurs solutions calculées dans la partie 2 afin de les comparer avec les solutions trouvées dans la partie 1, est, pour l'équation `AX = Y`, comme décrit ci-dessous :

- Première ligne : série de *tokens* représentant des nombres fractionnels, espacés les uns des autres, le tout représentant le vecteur de droite de l'équation matricielle, c'est à dire `Y`.
- Deuxième ligne : la matrice `A` enregistrée ligne par ligne, soit une ligne contenant `np` *tokens*. Les valeurs `n` et `p` sont facilement récupérables puisque on sait que `Y` contient toujours `n` valeurs.
- Troisième ligne : la solution particulière trouvée pour l'équation `AX = Y`, soit `p` *tokens* (le nombre de colonnes dans `A`).
- quatrième ligne : les vecteurs de la base du noyau de `A` trouvée, enregistrés les uns après les autres, soit `pq` *tokens*, avec `q = dim ker(A)` et bien sûr `p` est le nombre de composantes de la solution particulière enregistrée sur la troisième ligne (et aussi la dimension de l'espace vectoriel qui contient `ker(A)`, c'est-à-dire l'espace de départ (*domain*) de toute application linéaire que l'on pourrait associer à `A` ; canoniquement cela serait $\mathbb{R}^p$).
- Pour plus de lisibilité, j'ai ajouté des lignes commençant avec un #; ces lignes seront bien sûr éliminées dans le parsing de la partie 3.

Dans les cas limites :

- si aucune solution n'est trouvée (le système est insoluble, l'ensemble des solutions est `{}`), les lignes trois et quatre sont remplies chacunes par l'unique token `"NoSolution"` (en tant que chaîne de caractère).
- Si `dim ker(A) = 0`, la quatrième ligne sera remplie avec la chaîne `"NullVecSpace"`.

Comparaisons des solutions trouvées selon les deux méthodes :

Comme on va faire la vérification avec Sage, cela n'aurait vraiment aucun sens de vérifier la validité des solutions trouvés dans la partie 1.

Soit `AX = Y` l'équation matricielle du système étudié, `K` la familles de vecteur trouvée dans la partie 2 et qui devrait être des base du noyau de `A`, et `S` le vecteur correspondant à la solution particulière trouvée de même dans la partie 2. Appelons aussi `E` l'espace de départ de l'endomorphisme canoniquement associé à `A`.

Afin de vérifier que les solutions trouvées dans la partie 2 concordent avec celles de Sage, et sont valides, il faut vérifier :

- que les solutions particulières sont bien valides, c'est-à-dire, que `A*S == Y`. (Il a fallu transformer `S` en une matrice colonne.)
- que le sous-espace vectoriel engendré par l'hypothétique base `K` est bien égal au noyau trouvé par la commande sage `right_kernel`.
- que la cardinalité de la famille `K` est égale à la dimension de `ker(A)`.

L'unicité du noyau garantira alors l'unicité des solutions trouvées par les deux méthodes.

Notes sur la compatibilité Sage/Python

Sage n'a pas de problème à appeler des fonctions provenant de fichiers **.py**. Cependant, si une fonction écrite dans un fichier *Python* tente d'utiliser la fonction *built-in* `raw_input`, les choses se gâtent (plus précisément, *Sage* plante).

Pour y remédier, mais réutiliser un maximum de code, j'ai dû écrire des fonctions qui pourraient faire le travail tant pour la partie Python que pour les parties Sage. L'astuce a consisté à donner en paramètre une fonction qui sera utilisée à la place de `raw_input`. Plus précisément, *Sage* appelle une fonction *Python* avec sa propre version de `raw_input` en paramètre, et ainsi le code ne plante pas.

Le même problème s'est posé au niveau de la construction de nombres fractionnels : j'ai écrit un parseur qui fonctionne tant dans la Partie 1 que dans la Partie 2 (*ParsingPart12.py*). Plus précisément, c'est une fonction qui prend en paramètre un constructeur de fractions (`fractionMaker`), et qui renvoie un parseur, qui lui sera utilisé par une autre fonction qui crée un *prompt* (une interface utilisateur minimale) qui fonctionne tant pour les parties 1 et 2 que 3 (*UI.py*).

Remarques finales

- Il est arrivé que *Sage* affiche les résultats de la partie 1 (surtout pour le fichier *test2*)de manière légèrement dysfonctionnelle : parfois une partie de l'affichage était dupliqué, d'autres fois le `prompt` qui est censé se déclencher une fois l'affichage terminé, coupait en deux l'affichage…Le dysfonctionnement est aléatoire et ne dépend pas du code puisque deux tentatives l'une après l'autre sans modification du code résultait en un affichage différent. J'ai tenté de résoudre le problème avec `time.sleep` sans

grand succès. Comme je ne sais pas pourquoi ça plante, je n'ai pas pu résoudre le problème…

Il est *possible* que l'affichage soit meilleur lorsqu'on donne le fichier *test2* au prompt en premier, ou en tout cas tout seul, au lieu de le donner après les deux autres fichiers (*test0* et *test1*).

Aussi mentionnons que même quand l'affichage marche de manière normale, Sage met un temps démesuré pour afficher tous les résultats tous ensemble (au lieu de le faire au fur et à mesure). Cela a parfois mis entre 10 et 30 secondes pour s'afficher, alors ne perdez pas espoir trop vite !

(La version avec *Python* uniquement n'a heureusement et évidemment pas ce problème-là.)

- Les tests unitaires sont à lancer avec le fichier *UnitTests.py*, dans la partie 2.
- J'ai écrit une poignée de tests fonctionnels pour la partie 2: tester que pour des systèmes triviaux, la fonction `systemSolution` fait bien son job. Ces tests utilisent la même classe que les tests unitaires. Ils se trouvent dans le fichier *FunctionalTests.py*.