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.

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` le type de l'*output* de `f`.
- Si une fonction produit des effets de bords (*side effects*), on notera son type avec un point d’exclamation après la flèche, ainsi:

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

- En Python, une fonction peut très bien avoir des paramètres étant euxmême des fonctions (*cf* **Style de Programmation**). Par exemple voici le type de la fonction `map` :

```
map : (a -> b) . List a -> List 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 de cette manière :

  ```
  add : [Number 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`. `Number` représente une **classe de type** ou encore une **contrainte** 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`.

- Quelques types usuels et leur notation :

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

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

```
Matrix n = List (List n)
Vector n = List n
Family n = List (Vector n)
```

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)
```

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.

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/état extérieur n'a besoin d'être contrôlé ou simulé par peur de perturbations extérieures sur le fonctionnement d'une telle fonction, qui ne connaît que ce qui lui est donné en paramètres, et ne fait que renvoyer une nouvelle valeur de retour.

Il faut tout de même admettre qu'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, L):
    out = []
    for x in L:
        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.

Les outils les plus classiques des langages fonctionnels se révèlent naturellement utiles ici : 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 *output*.

`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)
```

Exemple d'utilisation dans mon code :

```
maybeSystemFromMatrix : [Number n] Matrix n . Vector n -> Maybe (System n)
echelonized           : [Number n] System n -> Maybe (System n)
normalized            : [Number n] System n -> Maybe (System n)
extractSolution       : [Number n] System n -> (Family n, Vector n)
```

```
maybeSolution = maybeSystemFromMatrix(matrix, rightSide
        ).maybeDo(echelonized, 0
        ).maybeDo(normalized
        ).maybeApply(extractSolution)
```

La première fonction, `maybeSystemFromMatrix` renverra ou bien `Just(system)` ou bien `Nothing`, en fonction de la validité des éléments donnés en paramètre. Par la suite, les fonction `echelonized`, `normalized` et `extractSolution` ne seront appelés que si tout s'est bien passé précédemment. Si à un quelconque moment, la valeur `Nothing` est produite, toute la séquence résultera en une valeur de `Nothing` automatiquement et aucune opération subséquente ne sera appliquée. `extractSolution` est une fonction qui n'échoue jamais, par conséquent elle est appliquée au résultat éventuel des opérations précédentes avec `maybeApply`. Au final, on obtient `maybeSolution : Maybe (Family n, Vector n)`.

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 t`, `String` étant le type du message d'erreur, et `t` 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`.

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)
```

Exemple d'utilisation de `Either` dans la fonction `eitherSequence` :

```
eitherSequence : (a -> Either e b) . List a -> Either e (List b)
def eitherSequence(f, xs):
    out = []
    for x in xs:
        y = f(x)
        if y.isLeft:
            return y
        else:
            out.append(y.rightValue)
    return Right(out)
```

Essentiellement, cette fonction agit comme une fonction `map : (a → b) . List a → List b` qui supporterait que la fonction donnée en entrée renvoie un message d'erreur au lieu d'une valeur de type `b`. Ainsi, `eitherSequence` renvoie *ou bien* le résultat du *mapping* de la liste d'entrée par la fonction d'entrée, *ou bien* le message correspondant à la première erreur rencontrée.

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 "pivotantes". A la fin du processus de échelonisation, et si tout s'est bien passé (pas d'équation `0 = n`, `n` non nul 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` le suivant :

- 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, soit `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 `q` *tokens*, `q` supérieur ou égal à `n`.
- quatrième ligne : les vecteurs de la base du noyau de `A` trouvée, enregistrés les uns après les autres, soit `qr` *tokens*, avec `r = dim ker(A)` et bien sûr `q` est le nombre de composantes de la solution particulière enregistrée sur la troisième ligne.
- cinquième ligne : ligne vide, qui sera de toute façon ignorée par le parser qui lira le fichier, pour plus de lisibilité.

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 `"NOTHING"` (en tant que chaîne de caractère).

Si `r = dim ker(A) = 0`, la quatrième ligne sera automatiquement laissée vide puique on aura tout simplement `qr = 0`.

Comparaisons des solutions trouvées selon les deux méthodes :

Soit `AX = Y` l'équation matricielle du système étudié, `B1` et `B2` les familles de vecteurs trouvées dans les parties 1 et 2 et qui devraientt être des base du noyau de `A`, et `S1` et `S2` les vecteur correspondant aux solutions particulières trouvées de même dans les parties 1 et 2.

Afin de vérifier que les solutions concordent et sont valides, il faut vérifier :

- que les solutions particulières sont bien valides, c'est-à-dire, que `A*S1 == Y` et `A*S2 == Y`
- que les hypothétiques bases sont libres et de cardinalité identiques à la dimension du noyau trouvé par Sage, et de plus que pour chaque vecteur `u` dans `B1` et `B2`, on ait `A*u = 0`, c'est-à-dire, que `u` est bien dans `ker(A)`.

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