TP Haskell #6/7 (CS 222)

Grenoble INP – Esisar – année 2022-23

Durée : 1h30
Consignes et date de rendu : Voir Chamilo

Objectifs :

1. Obtenir l'intuition d'une monade comme un nouvel environnement d'exécution.
2. Utiliser les opérateurs `fmap` (`<$>`) et `bind` (`>>=`) dans des cas simples.

Mise en place du TP

Réalisez le TP dans un répertoire de travail `CS222/TP6_VotreNom`. Chaque exercice `XXX.hs` sera accompagné d'un script GHCi, nommé `test_XXX.ghci`, contenant les expressions sur lesquelles vous avez testé votre code, et exécutable de la façon suivante dans GHCi :

```
:load XXX.hs
[... affichage terminant par Ok, one module loaded.]
:script test_XXX.ghci
[... affichage de l'évaluation de vos tests]
```

Consignes communes pour l'écriture des fonctions Haskell :

- Chaque définition de fonction sera précédée de sa déclaration de type.
- Privilégier la réutilisation des fonctions écrites précédemment.
- Privilégier le filtrage de motif (*pattern-matching*) à du code conditionnel.
- N'hésitez pas à utiliser les fonctions de la bibliothèque standard de Haskell.

Exercice 1 — Propagation automatique des erreurs

Fichiers à rendre : `Erreurs.hs`, `test_Erreurs.ghci`.

Une des monades les plus courantes est la monade associée au type `Maybe`, qui permet de gérer des erreurs. L'objectif de cet exercice est de manipuler des fonctions qui peuvent échouer, et de montrer que les monades en simplifient l'usage au point où on peut s'en servir sans se préoccuper constamment des erreurs.

On se donne la liste des planètes du système solaire, accompagnées de leur distance au Soleil moyenne en Unités Astronomiques (UA), dans l'ordre croissant.

```haskell
planetes :: [(Double, String)]
planetes = [
  (0.39, "Mercure"),    (0.72, "Venus"),      (1.00, "Terre"),
  (1.52, "Mars"),       (5.20, "Jupiter"),    (9.54, "Saturne"),
  (19.2, "Uranus"),     (30.1, "Neptune")]
```

- Écrire une fonction `distance :: String -> Maybe Double` qui prend en paramètre un nom de planète et renvoie sa distance au Soleil. Si le nom demandé n'est pas présent dans la liste, la fonction renverra `Nothing`. *(Vous aurez sans doute besoin d'une sous-fonction qui prend en paramètre la liste dans laquelle on cherche.)*

- Écrire une fonction `suivante :: Double -> Maybe String` qui prend en paramètre une distance `d` en UA et renvoie le nom de la première planète dont la distance au Soleil est strictement supérieure à `d`. S'il n'y en a pas, la fonction renverra `Nothing`. Il vous est conseillé :
  1. D'utiliser `filter` pour obtenir la liste des planètes à distance strictement supérieure à `d` du Soleil ;
  2. Dans le cas où la liste est vide, renvoyer `Nothing`, sinon renvoyer le nom de la première planète.
- Écrire une fonction `ua_vers_km :: Double -> Double` qui convertit une distance d'UA en km (multiplication par `1.496e+8`).

Nous avons maintenant deux fonctions dont le type est de la forme `a -> Maybe b`, que l'on peut comprendre intuitivement comme : "une fonction de type `a -> b` mais qui peut échouer". C'est un bon début, mais on ne peut pas facilement les composer. En effet, une fonction `f :: a -> Maybe b` ne peut être composée ni avec une fonction `g :: b -> c` ni avec une fonction `h :: b -> Maybe c`, puisque `b` et `Maybe b` sont deux types différents. Plus concrètement, *on ne peut pas continuer le calcul avec `g` ou `h` tant qu'on n'a pas vérifié si `f` a renvoyé une erreur.*

Cas où l'on continue avec une fonction qui ne peut pas échouer

- Écrire une fonction `distance_km_1 :: String -> Maybe Double` qui associe à un nom de planète sa distance au Soleil en km, en appelant `distance` puis `ua_vers_km`.

Remarquez que vous avez besoin de filtrer la valeur renvoyée par `distance_planete` pour déterminer si une erreur s'est produite ; c'est un peu fastidieux.

- Écrire une fonction `fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b` qui prend en argument une fonction et une valeur de type `Maybe a`. Elle renverra `Nothing` si la valeur est `Nothing` et appliquera la fonction sous le `Just` sinon.

Cette fonction est disponible dans la bibliothèque standard ; vous l'appelerez par son nom `fmap` ou via l'opérateur infixe `<$>`. Intuitivement, `<$>` *applique une fonction dans le cas `Just` et ne fait rien dans le cas `Nothing`.*

- En déduire une version améliorée `distance_km_2 :: String -> Maybe Double` qui ne fait pas de filtrage de motif.

Cas où l'on continue avec une fonction qui peut échouer

- Écrire une fonction `distance_suivante_1 :: String -> Maybe Double` qui prend en paramètre le nom d'une planète et renvoie la distance au Soleil de la planète qui la suit. Par exemple, sur l'entrée `"Terre"` la fonction renverra la distance entre le Soleil et Mars.

  Pour cette fonction, vous devrez appeler `distance`, puis `suivante`, et de nouveau `distance`. Ces appels pouvant échouer, il vous est conseillé d'inspecter chaque valeur de retour à l'aide d'une expression `case`, comme ceci :

  ```
  case distance p of
    Nothing -> ... {- erreur dans le calcul de la distance -}
    Just d  -> ... {- la distance est d -}
  ```

  Si vous ne l'avez pas encore fait, ajoutez des tests dans `test_erreurs.ghci`.

La fonction précédente est un bon début, mais ça fait beaucoup de code très répétitif pour propager le `Nothing`. Comme précédemment, on veut donc écrire une fonction auxiliaire qui ferait ça à notre place.

- Expliquer pourquoi l'opérateur `<$>` (`fmap`) n'est pas suffisant pour cette tâche. Vous pourrez analyser les types des fonctions et comparer aux fonctions `f`, `g` et `h` de l'introduction.
- Écrire une fonction `bind_Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b` qui renvoie `Nothing` sur l'entrée `Nothing` et appelle la fonction fournie sur l'entrée `Just a`. Expliquer la différence entre `fmap` et `bind_Maybe` en termes de capacité à renvoyer des erreurs.

Cette fonction est disponible dans la bibliothèque standard ; vous l'appelerez par son nom `bind` ou via l'opérateur infixe `>>=`. Intuitivement, `>>=` sert à *poursuivre un calcul en restant dans le monde où les fonctions peuvent renvoyer des erreurs.*

- Écrire une version améliorée `distance_suivante_2 :: String -> Maybe Double` qui utilise `>>=` et des lambdas à la place des `case`.

On rappelle qu'en Haskell un bloc `do` peut être utilisé pour enchaîner plusieurs calculs connectés par des bind (`>>=`). La syntaxe est la suivante :

```haskell
do x <- un_calcul
   y <- un_autre_calcul
   valeur_retour

-- Équivalent à:
un_calcul >>= (\x ->
  un_autre_calcul >>= (\y ->
    valeur_retour))
```

- Écrire une dernière version `distance_suivante_3 :: String -> Maybe Double` qui utilise un bloc `do`. Remarquez que le code ne fait plus du tout de filtrage : les erreurs sont propagées automatiquement !

Exercice 2 — Gestion des entrées/sorties

Fichiers à rendre : `IO.hs`, `test_IO.ghci`.
Les concepts de cet exercice seront utilisés dans le programme d'ordonnancement de tâches.

Avec les monades, nous pouvons enfin aborder les entrées-sorties. Leur existence semble contre-intuitive car Haskell est un langage pur (sans effets de bord). Cette pureté a plein d'avantages ; par exemple, deux appels de fonction identiques renvoient forcément la même valeur, et on peut calculer les expressions dans l'ordre qu'on veut. Pourtant deux appels à `scanf()` en C ne renvoient pas forcément la même valeur, et changer l'ordre de deux appels à `printf()` change le comportement du programme. Alors comment concilier ces deux aspects ?

La monade des entrées-sorties, `IO`, résoud ce problème. L'idée majeure c'est qu'une fonction comme `print :: Show a => a -> IO ()` qui affiche une valeur dans le terminal, n'affiche en fait pas la valeur, mais renvoie une *action* qui peut être exécutée plus tard. Une valeur de type `IO T` c'est donc une action qui, quand elle est exécutée, fait des entrées-sorties puis renvoie une valeur de type `T`.

Les actions restent non exécutées jusqu'à ce que le code soit lancé dans GHCi. Vous avez déjà vu que si vous tapez une expression comme `1+2`, GHCi vous affiche leur valeur. En plus de ce comportement, si vous saisisez une action comme `print (1+2)` dont le type est `IO T`, alors GHCi exécute l'action puis affiche le résultat.

- Écrire une fonction `afficher_avec_etoiles :: Show a => a -> IO ()` qui utilise `show` et `print` pour afficher une valeur avec trois étoiles de chaque côté. Testez-la ensuite dans GHCi comme suit :

```
ghci> afficher_avec_etoiles 3.14
***3.14***
```

Ajoutez dans `test_io.ghci` un test similaire.

- Écrire une fonction `echo :: IO ()` qui lit une chaîne de caractères dans le terminal avec `getLine :: IO String` puis l'affiche avec `putStrLn :: String -> IO ()`. Contrairement au `Maybe` de l'exercice précédent vous ne pouvez pas filtrer une valeur de type `IO ()`, vous devez utiliser `bind` (`>>=`) ou un bloc `do`.

On veut maintenant écrire une fonction similaire au programme Unix `wc` (*word count*), qui compte le nombre de lignes, mots et octets dans un fichier.

- Écrire une fonction `wc :: String -> IO ()` qui prend en paramètre le nom d'un fichier et affiche ses statistiques au format suivant :

  `<nombre de lignes> <nombre de mots> <nombre d'octets> <nom du fichier>`

  Par exemple:

  `22 96 518 IO.hs`

  On utilisera la fonction `readFile :: FilePath -> IO String` (où `FilePath` est un alias de `String`) pour obtenir les contenus du fichier. Ajoutez de plus `import Data.List` au début de votre fichier pour accéder aux fonctions `words` et `lines` qui génèrent la liste des mots et lignes d'une chaîne de caractères, respectivement.

- Ajouter une fonction `main :: IO ()` qui récupère les arguments de ligne de commande avec `getArgs :: IO [String]` (disponible après `import System.Environment`) et appelle `wc`.

De cette façon, vous pouvez compiler un exécutable et l'appeler comme le `wc` classique d'Unix.

```
% ghc --make IO.hs
(...)
% ./IO IO.hs
22 96 518 IO.hs
```


Exercice 3 — Monade des listes et problème du cavalier

Cet exercice est facultatif ; vous pourrez rendre les fichiers `Cavalier.hs` et `test_cavalier.hs`.

Dans le premier exercice, on a vu comment la monade `Maybe` modélise le comportement des programmes qui peuvent échouer. Une autre monade intéressante est la monade des listes, qui modélise des fonctions *non-déterministes* (ie. des fonctions qui peuvent faire des *choix*). Une fonction `a -> b` mais qui est non-déterministe sera codée par une fonction `a -> [b]` ; la liste représente toutes les valeurs possibles que la fonction peut renvoyer à l'issue de ses choix (qui sont invisibles quand on l'appelle).

Ces fonctions sont notamment utiles pour résoudre les problèmes *d'exploration* comme le problème du cavalier. Dans ce problème, un cavalier est placé sur un échiquier de m lignes et n colonnes. Le cavalier se déplace « en L » comme illustré ci-dessous. La question est la suivante : est-il possible de déplacer successivement le cavalier pour lui faire visiter toutes les cases de l'échiquier sans passer deux fois par la même case ?

Un programme pour résoudre ce problème pourra *explorer* les différentes options de déplacement du cavalier à chaque étape du jeu. Généralement il y en a plusieurs, chacune donnant un résultat différent, et offrant plusieurs nouvelles options. Un programme non-déterministe utile dans cette situation est

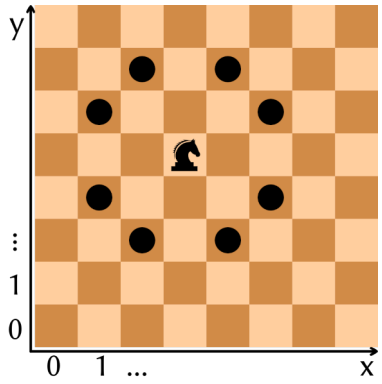

Figure 1: Mouvement du cavalier et coordonnées sur l'échiquier.

celui qui *choisit* un déplacement à chaque étape, et continue jusqu'à trouver une solution ou tomber à court d'options. Pour savoir s'il y a une solution, il suffit de regarder toutes les exécutions possibles du programme non-déterministe, et la monade des listes en Haskell nous aide à calculer ça.

On vous fournit le fichier `ProblemeCavalier.hs` modélisant le problème pour un échiquier de taille 4×3 ; les quelques fonctions fournies sont expliquées dans les commentaires. (Il ne vous est pas demandé de comprendre le code, simplement d'utiliser les fonctions.) Dans `Cavalier.hs` vous ajouterez `import ProblemeCavalier` pour utiliser le module.

- Écrire une fonction `mouvements :: Etat -> [Position]` qui sur l'entrée `(p, (x, y))` indique toutes les cases que le cavalier placé en `(x, y)` peut atteindre. Il y a 8 mouvements en L possibles, mais vous ne devez garder que les cases qui sont dans les bornes de l'échiquier et pas déjà visitées dans `p`.
- En déduire une fonction `etats_suivants :: Etat -> [Etat]` qui liste tous les états possibles du jeu après un mouvement du cavalier. On notera que si le cavalier est bloqué (et donc la partie perdue) la liste renvoyée est vide.

On peut voir `etats_suivants` comme une fonction non-déterministe qui *choisit* un mouvement possible et renvoie l'état correspondant. L'opérateur `>>=` (même notation que pour `Maybe`) nous permet dans la suite de continuer ce calcul avec une autre fonction non-déterministe et ainsi d'enchaîner les choix. Dans ce monde non-déterministe, `l >>= g` est la liste de toutes les valeurs que `g` peut renvoyer (en fonction de ses choix internes) après avoir été appelée sur n'importe laquelle des valeurs de `l`.

- Écrire une fonction `explorer_etats :: Etat -> [Etat]` qui détermine tous les états gagnants accessibles à partir de l'état fourni.
  - Si l'état fourni est gagnant (ie. `plateau_plein p` est vrai), alors on renverra une liste contenant juste cet état ; la recherche s'arrête.
  - Sinon, on calculera les états suivants (dans la vision non-déterministe, on *choisira* un de ces états) et on utilisera un appel récursif pour continuer l'exploration.
- En déduire une fonction `etats_gagnants :: Position -> [Etat]` qui liste tous les états gagnants quand le cavalier commence à la position indiquée.
- Calculer la liste `positions_initiales_realisables :: [Position]` de toutes les positions initiales à partir desquelles le cavalier *peut* parcourir tout l'échiquier en passant exactement une fois par chaque case. Il vous est conseillé d'utiliser une compréhension de listes pour énumérer les coordonnées initiales.

Exercice 4 — Dessin du dragon de Heighway avec les entrées/sorties

Cet exercice exploratoire est facultatif ; vous pourrez rendre `Dragon.hs` et `test_Dragon.ghci`.

On vous fournit le module `Pics` contenant une infrastructure de dessin utilisant la bibliothèque Haskell SVG. (Vous pouvez l'installer avec `cabal install svg-builder --lib` sur vos machines perso, en espérant que la librairie existe en salle de TP.)

- Lire et prendre en main le code fourni. Dessiner des choses dans `test_Dragon.hs`, par exemple des séries de maisons. :)

Vous pouvez générer un fichier SVG à partir d'une `Picture` avec la fonction `output` de `Pics.hs`. Pour visualiser, vous pouvez utiliser ImageMagick pour faire un rendu JPG :

```
% convert image.svg image.jpg
```

- Pour comprendre le code fourni, on pourra se reporter à doc wiki qui explique la structure mathématique simple de monoïde, et on pourra répondre aux questions suivantes (GHCi est votre ami):
  - quels dessins sont codés par le type de données `Picture`?
  - que fait la fonction `polyline`?
  - que fait `(°/)`?
  - que fait `rotateP`?
  - à quoi servent les arguments de `mkLandscape` ?
  - (il n'est pas nécessaire de comprendre toSvg)
- Écrire une fonction récursive `dragon :: Int -> Float -> Picture` qui calcule un dragon de Heighway. Le premier argument désigne le nombre d'appels récursifs, le deuxième la taille de chaque segment.

On pourra utiliser une fonction auxiliaire qui retourne non seulement la figure courante mais aussi le dernier point calculé.

- Générez un dragon avec 14 appels récursifs et des segments de longueur 8 pour obtenir la figure ci-dessous.

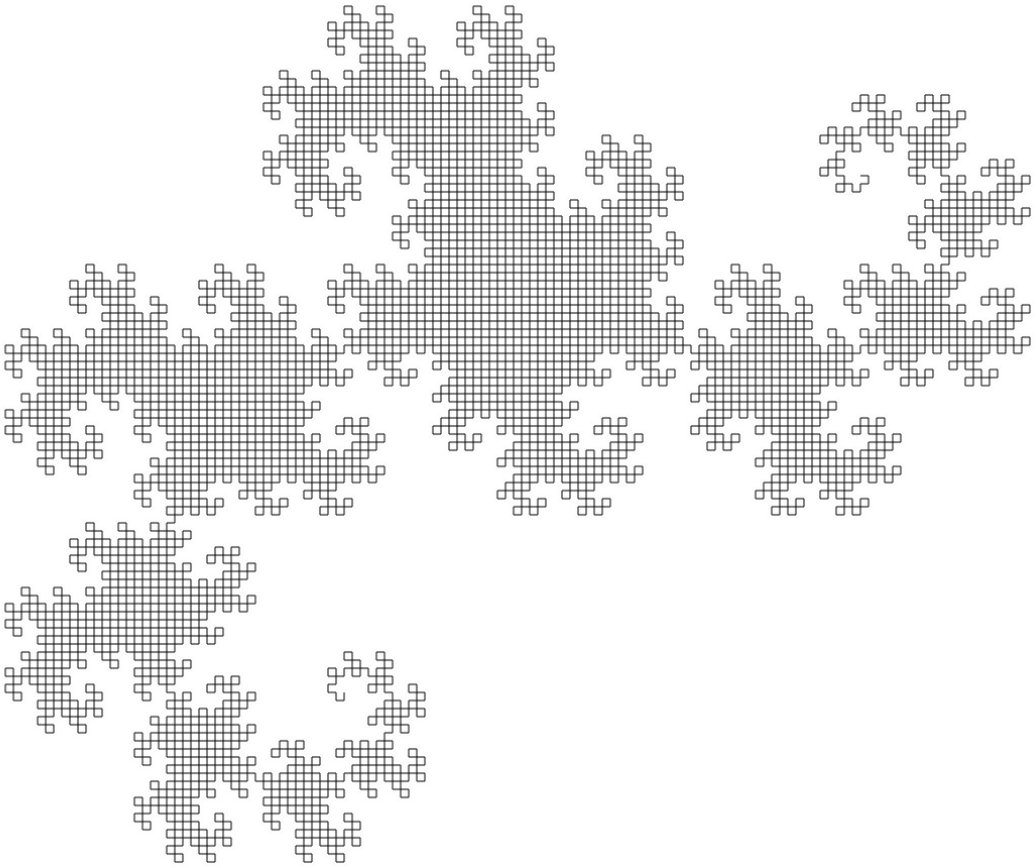

Figure 2: Dragon de Heighway