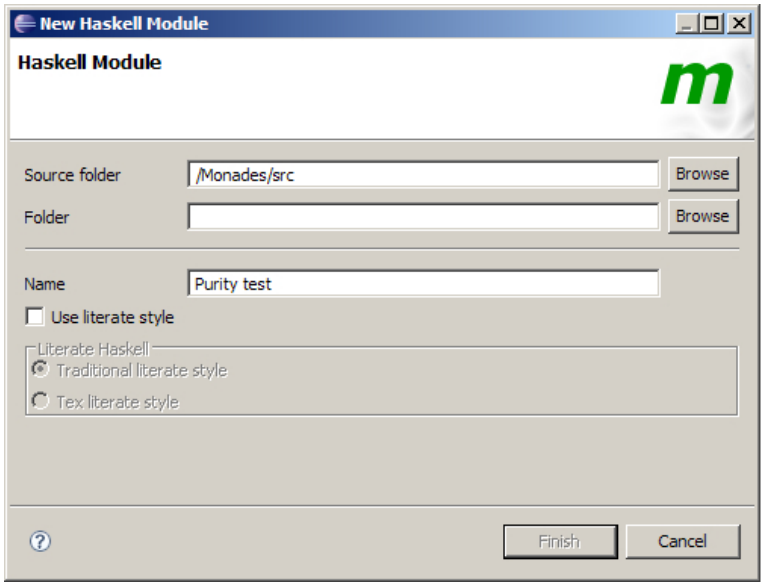
www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Le paradoxe de la pureté en programmation fonctionnelle : les monades salvatrices

Michaël Monerau

Janvier 2009



Introduction

Il existe essentiellement deux types de langages de programmation : les langages dits *impératifs* et les *fonctionnels*.

Les premiers tirent leur nom du fait que le programmeur a accès (plus ou moins) directement à la suite d'instructions que son programme va générer, par exemple par l'utilisation de boucles ou en contrôlant directement l'état de la mémoire. Historiquement, ce sont les premiers à êtres apparus. Pour fixer les idées, on peut citer `C` ou `C++` comme les deux langages impératifs les plus utilisés.

Il a fallu attendre la fin des années 70 pour que des chercheurs s'intéressent à un nouveau paradigme : la *programmation fonctionnelle*. Dans ce monde-là, on oublie les variables mutables et les seuls objets manipulés sont des fonctions (éventuellement partielles) et des constantes. Cela est directement inspiré du lambda-calcul dans lequel on applique et construit des fonctions partielles ou non.

Si la programmation fonctionnelle est restée jusque-là presqu'exclusivement académique, elle commence à connaître un essor réel. Mais elle bouleverse complètement les habitudes de la programmation impérative et demande ainsi un effort certain d'apprentissage pour être efficacement utilisée.

Dans ce travail, nous essaierons de comprendre comment on peut utiliser ce cadre *a priori* rigide de la programmation fonctionnelle pour en tirer le meilleur parti. Afin de briser le suspens, mentionnons d'ores et déjà que nous userons et abuserons d'une notion venue de la théorie des catégories : la monade. Dans un premier temps, nous étudierons le paradigme de la programmation fonctionnelle et utiliserons le langage Haskell pour illustrer nos propos. Puis nous essaierons de simuler quelques comportements qu'on pourrait penser de nature impérative en Haskell. Enfin, nous généraliserons ces techniques à l'aide des monades pour obtenir une structure de programmation qui bénéficiera à la fois de la robustesse de la programmation fonctionnelle et de la souplesse de la programmation impérative.

Plan

1 Programmation fonctionnelle & Haskell

1.1 Les principes de base de la programmation fonctionnelle

Comme on l'a dit en introduction, la programmation fonctionnelle refuse toute notion de variable non-constante (*mutable*) et d'effets de bord : les seuls objets manipulés sont des fonctions ou des constantes : on ne connaît pas l'assignation. Les fonctions peuvent être passées (partiellement ou non) en argument à d'autres fonctions, ou même être retournées comme valeur. Cela donne souvent une formulation très compacte et générique d'algorithmes qui seraient fastidieux à exprimer en programmation impérative.

La fonction `map` en est un bon exemple : on appelle `map` avec en arguments une fonction f et une liste $[a_1, \ldots, a_n]$, et on obtient en sortie la liste des résultats $[f(a_1), \ldots, f(a_n)]$.

Une des implications importante de ce genre de fonctions, dites *pures* est que leur absence d'effet de bord fait que leur exécution ne dépend aucunement de leur environnement d'exécution. Si on appelle une fonction plusieurs fois sur le même argument, elle rendra *toujours* la même réponse quelque soit l'endroit depuis lequel elle est appelée ou l'état de l'environnement d'exécution. Ainsi, si on prouve qu'une fonction pure réalise bien ce qu'on veut, on peut être certain qu'elle n'engendrera jamais aucun bug.

C'est bien sûr un fait très important et cela justifie en partie tout l'intérêt qu'on prête à ce genre de paradigme de programmation : la modularité permet de prouver globalement qu'une partie d'un programme ne *peut pas* bugger, puisque chaque élément est correct et est garanti de l'être en toute circonstance.

De plus, ce genre d'invariant sur le comportement des fonctions pures, et en particulier leur indépendance d'exécution, peut permettre au compilateur de nombreuses optimisations parmi lesquelles la parallélisation trouve bonne place.

Cependant, afin d'être plus souples d'utilisation, certains langages fonctionnels autorisent malgré tout l'usage de variables mutables, de boucles, et plus généralement de structures de programmation impérative. Ces langages sont dits fonctionnels *impurs* à l'opposé des autres qui sont *purs* et qui ne proposent aucun des outils impératifs classiques. Parmi eux, on peut par exemple citer OCaml qui autorise notamment les variables locales sous la forme de `ref`.

Une des autres particularités partagées par beaucoup de langages fonctionnels est l'inférence de type. Lorsqu'elle est définie, une fonction reçoit le type le plus général qui la représente. Par exemple, la fonction *identité* reçoit le type `a -> a`, où `a` désigne n'importe quel type, et la fonction `x` $\mapsto$ `x mod n` reçoit le type `Int -> Int`.

Je ne m'attarderai pas sur ces notions, je suppose que le lecteur a un minimum de familiarité avec le sujet.

1.2 Haskell

Dans la suite de cette étude, nous utiliserons le langage **Haskell** pour illustrer nos propos. C'est un langage purement fonctionnel né dans le milieu des années 80 justement de l'idée de regrouper sous une seule bannière les dernières avancées en date de la théorie des langages de programmation. La dernière révision du standard date de 1998. Ce langage est en plein essor et possède une communauté très active.

Haskell se place donc du côté des langages purement fonctionnels. Il en possède les caractéristiques classiques sur lesquelles ne ne reviendrons pas. La syntaxe est habituelle de ce genre de langages et le lecteur se laissera guider sans crainte. On peut tout de même dire qu'on note `x : : t` pour dire que `x` est de type `t`.

Essentiellement, on peut citer comme particularités d'Haskell un *pattern matching* extrêmement pratique qui permet une programmation sous forme équationnelle, une inférence de type très efficace qui permet un polymorphisme très général associé à un système de "Typeclass" particulièrement pratique sur lequel nous allons nous attarder quelque peu.

Le principe de *typeclass* permet de capturer les propriétés du type d'un objet. Souvent, pour généraliser au maximum le type d'une fonction, il faut tout de même faire une assomption sur le type de ses arguments. Par exemple, supposons qu'on veuille écrire la fonction suivante :

```
exemple_egalite x y = if (x == y) then 1 else 0
```

Quel type donnerait-on à cette fonction `exemple_egalite` ? On est tenté de dire que :

```
exemple_egalite :: a -> a -> Int
```

où `a` est un type généralisé. Mais ce n'est pas vrai : l'égalité ne peut pas être testée entre deux fonctions par exemple (un résultat théorique nous assure même que cette question est indécidable en général). Ainsi, il faut restreindre le polymorphisme aux types `a` pour lesquels l'égalité a un sens défini.

On définit alors la *typeclass* notée `Eq` des types qui proposent la méthode de comparaison `==`, et le type de `exemple_egalite` est alors :

```
exemple_egalite :: (Eq a) => a -> a -> Int
```

De même, on peut observer que les types des opérateurs de base sont :

```
(+) :: (Num a) => a -> a -> a
(-) :: (Num a) => a -> a -> a
(*) :: (Num a) => a -> a -> a
(+) :: (Fractional a) => a -> a -> a
```

où `Num` désigne la *typeclass* des nombres en général, et `Fractional` celle des nombres qui peuvent être exprimés sous forme de fractions (`Int` n'est donc pas une instance de `Fractional`).

Comme on le voit, ce système permet un polymorphisme plus souple que celui qu'on peut trouver en OCaml par exemple, où l'opérateur d'addition a fatalement pour type `int -> int -> int`.

1.3 Où mène la pureté ?

Il est temps de se demander si tout cela mène bien à quelque chose. Cette pureté est élégante et théoriquement très satisfaisante, mais comment *faire* effectivement quelque chose sans boucles ni variables ? C'est le paradoxe de la programmation fonctionnelle pure. Le code qu'on écrit est totalement exempt de tout effet de bord et pourtant on peut bien programmer ce qu'on veut. Comment ?

C'est ici que vont intervenir les monades. Elles sont en quelque sorte le pont entre le monde réel impur et le monde de la pureté fonctionnelle. C'est au sein de ces monades, et seulement à ces endroits, qu'on pourra avoir des comportements de type impurs comme l'intéraction (indispensable évidemment) avec le monde extérieur (entrées / sorties) ou des effets de bord lors de l'exécution de fonctions.

Mais tout ne s'écroule pas : l'impureté est canalisée. En effet, le programmeur peut lire simplement sur le type d'une fonction si elle est pure ou impure. Et dans ce dernier cas, selon la monade mise en jeu, il peut même savoir quels types de comportements impurs la fonctions peut présenter (entrée / sortie, écriture de fichiers, intéraction avec l'utilisateur, appels externes, effets de bord).

Ainsi, l'approche pure n'est pas baffouée, et elle nous mène même sur des terres plus fertiles qu'on ne l'imaginait de prime abord. Dans les parties pures du programme, on peut toujours appliquer notre paradigme de la vérification du comportement indépendamment de l'extérieur, et de l'optimisation automatique. Cependant, dès lors que l'on voit l'apparition d'une monade dans le type d'une fonction, alors on sait qu'on peut être entrain d'avoir affaire à des comportements impurs (même si ce n'est pas nécessairement le cas, une monade peut tout à fait être utilisée dans un cadre pur).

2 Premiers pas vers la pure impureté

2.1 Un analogue de `let` dans l'esprit fonctionnel

En Haskell, chaque fonction est constituée d'une seule expression. Il n'y a pas d'équivalent syntaxique au point-virgule de `C` par exemple. Et cela fait sens : si on avait "`ligne 1 ; ligne 2`", que ferait la ligne 1 ? Par définition, sa valeur de retour ne serait pas utilisée. Donc sa seule action possible est par effet de bord. Donc cette structure n'est pas autorisée dans du code pur.

Malgré tout, tous nos programmes ne sont pas que des enchaînements de *one-liners* (petites fonctions qui ne comportent qu'une seule ligne). Ou plutôt, on n'a pas envie de les écrire comme tels pour des raisons d'esthétique et de facilité de relecture. En effet, il est équivalent d'écrire :

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com


```
-- (Ce qui suit deux tirets est un commentaire en Haskell)

f :: Int -> Int -> Int -- par exemple
f x y = (g x (x+y)) + (h (y*x) y) -- où g et h sont des fonctions cohérentes

-- ou définition équivalente :
f :: Int -> Int -> Int
f x y =
  let resultatG = g x (x+y)
      resultatH = h (y*x) y
      in
      resultatG + resultatH
```

Il est bien évident que la deuxième version est plus claire, et plus facilement débuggable si besoin est. L'utilisation de l'opérateur `let` en Haskell est bien autorisé. Ce n'est pas contradictoire avec la pureté de `f` : les valeurs assignées à `resultat{F,G}` ne sont pas mutables, et donc on peut bien considérer que c'est juste une convention syntaxique et qu'on pourrait remplacer leur expression dans la ligne d'addition pour retomber sur la première version de `f`. Cependant, cette utilisation de `let` ne nous satisfait pas vraiment car elle ne se généralise pas bien. Essayons de trouver quelque chose d'équivalent mais qui serait plus dans l'esprit fonctionnel.

À présent, il nous faut trouver un moyen syntaxique de diviser une expression en plusieurs, en restant dans le cadre sémantique du calcul pur de Haskell.

Définissons l'opérateur suivant (entre parenthèses pour dire qu'il est infixe) :

```
(>>=) :: Int -> Int -> a -> a
x >>= f = f x
```

Ceci est tout à fait trivial, et pourtant, grâce à la notation de lambda-fonction offerte par Haskell (`\x -> e` équivaut à $\lambda x.e$ en lambda-calcul), nous allons pouvoir utiliser `>>=` de manière très efficace. En effet, si on écrit :

```
-- l'expression globale est de type Bool
3 >>= \x -> if (x == 20) then True else False
```

On obtient quelque chose qui n'est pas innocent : la deuxième partie de l'expression (après le `->`) pourrait très bien se trouver après un bloc "`let x = 3 in`". Il suffit de réorganiser le code comme suit pour que l'astuce ne paraisse pas si anodine :

```
3 >>= \x ->
if (x == 20) then True else False
```

Il se trouve que Haskell définit une syntaxe équivalente à la prédente qui est exactement ce qu'on cherchait (en réalité, ce qui suit ne compile pas car il faudrait se placer dans une monade, mais passons sous silence cela à des fins didactiques) :

```
x <- 3 -- syntaxiquement équivalent à "3 >>= \x ->"
if (x == 20) then True else False
```

Et l'exemple de tout à l'heure, même s'il comporte deux variables, s'écrit donc très naturellement en faisant cette fois-ci deux applications de `>>=` (même remarque) :

```
f :: Int -> Int -> Int
f x y =
  resultatG <- g x (x+y)
  resultatH <- h (y*x) y
    resultatG + resultatH
```

2.2 Poussons l'analogie : composition d'actions et pérennité

On sent bien que ce qu'on vient de définir pour imiter le comportement de `let` va pouvoir se généraliser. Déjà, on peut remplacer `Int` par un autre type. Mais outre cela, on peut réutiliser la notion de "composition de ligne" par l'utilisation de fonctions partielles et d'une opération de composition moins triviale.

Il faut remarquer que dans l'exemple précédent, la fonction de composition `>>=` ne faisait que transmettre l'information à la ligne suivante, agissant seulement comme une sorte de glue.

On pourrait imaginer un comportement plus complexe où le passage d'une ligne à l'autre servirait, outre transmettre l'information de la valeur de la ligne passée, à faire remonter de l'information de la ligne qui vient. Car en effet, au final, c'est bien simplement la fonction `>>=` qui est évaluée et on n'est pas dans une stratégie d'exécution de la première ligne sans jamais y revenir.

Pour rendre cette idée plus concrète, nous allons construire un "logger". C'est-à-dire que nous voulons pouvoir exécuter plusieurs lignes de code à la suite en pouvant à chaque fois enregistrer un texte qui pourrait par exemple être affiché à l'écran.

Nous définissons donc une nouvelle version de l'opérateur `>>=` :

```
-- [String] est une liste de chaines de caractères, ie. la liste des lignes de log ici
-- (a, b) est le type couple (a, b)
(>>=) :: (Int, [String]) -> (Int -> (Int, [String])) -> (Int, [String])
(x, str) >>= f =
    let (y, str') = f x   in      -- utilisons let pour plus de clarté
    (y, str ++ str')              -- ++ est la concaténation
```

Que fait cette fois-ci le combinateur ? Il prend l'entrée `x` qui est la valeur sur laquelle faire le calcul, et il prend l'état actuel du buffer de log. Il lance ensuite le calcul à l'aide de la fonction `f : : Int -> (Int, [String])`. Ce calcul renvoie la nouvelle valeur `y`, et aussi un log qui a été généré lors du calcul, enregistré dans la variable pure `str'`. Enfin, le combinateur décide de renvoyer comme combinaison de ces deux opérations la valeur finale du calcul `y` et la valeur finale du buffer de log qui est ce qui avait déjà été écrit (`str`) ainsi que ce qui vient d'être dit lors du nouveau calcul (`str'`).

Et alors, on a atteint un bel objectif. Voyons cela en images.

```
(3, [])                  >>= \x ->
(x + 3, "j'ai ajouté 3") >>= \y ->
(y + x, "j'ai ajouté par x et y")
```

Cette expression s'évalue en `(9, ["j'ai ajouté 3", "j'ai ajouté x et y"])`. Et plus étonnant encore :

```
action :: Int -> (Int, [String])
action x = (x+1, "Une action va avoir lieu")

(3, [])                  >>=
action                   >>= \x ->
(x + 3, "j'ai ajouté 3") >>=
action                   >>= \y ->
(x * y, "j'ai multiplié x par y")
```

Et on obtient maintenant `(28, ["Une action va avoir lieu", "j'ai ajouté 3", "Une action va avoir lieu", "j'ai multiplié x par y"])`.

Alors, que se passe-t-il ? On a réussi à faire un effet de bord avec seulement du code pur puisque le comportement interne de la fonction `action` influe sur ce qui se passe à l'extérieur ! Mais non, il n'y a pas d'effet de bord. Le comportement de `action` est bien toujours le même quelque soit son environnement : sur la même entrée on obtient toujours la même sortie.

Simplement, on a trouvé un moyen pour transmettre de l'information d'un endroit vers un autre : la fonction `action` peut faire un log de ses actions (qui pourraient varier si jamais on mettait une condition dépendant de l'entrée/sortie dans son corps par exemple).

Même si c'est surprenant la première fois qu'on y est confronté, il faut garder à l'esprit que malgré l'apparent découpage en lignes, le code qui précède n'est *qu'une seule expression* où l'opérateur `>>=` n'est qu'une fonction infixe appelée en cascade et avec plusieurs arguments. C'est la composition de ces fonctions qui permet de donner l'illusion du découpage qu'on aurait en style impératif.

2.3 Les monades comme généralisation

Brisons dès maintenant le suspens : ce que nous venons de construire est une monade dissimulée. Nous allons définir le cadre formel de la monade à présent, et nous verrons que ce qui précède n'en est qu'un cas très particulier.

C'est d'ailleurs ainsi qu'est apparue l'idée d'introduire les monades en calcul fonctionnel. Au départ, les programmeurs avaient mis au point des modèles de ce type-là pour simuler des comportements impératifs connus. C'est alors que, comme le dit Wadler dans [Wad97], l'idée le cadre formel des monades a uniformisé toutes les astuces qui avaient été trouvées jusque là. Cette avancée est due en grande partie à Moggi et Kleisli pour leur travail (*a priori* sans lien !) sur les catégories.

Nous allons commencer par définir ce qu'on entend par monade en théorie des catégories, puis nous verrons comment on peut s'en servir en Haskell.

2.3.1 Une monade du point de vue catégorique

La théorie des catégories nous donne une définition très précise de ce qu'est une monade :

[2.A] Définition ***(Monade)***

Soit $\mathcal{C}$ une catégorie. Une monade sur $\mathcal{C}$ est un triplet $\langle T, \eta, \mu \rangle$ où $T : \mathcal{C} \longrightarrow \mathcal{C}$ est un foncteur, et où $\eta : Id_{\mathcal{C}} \stackrel{\cdot}{\longrightarrow} T$ et $\mu : T^2 \stackrel{\cdot}{\longrightarrow} T$ sont des transformations naturelles. η est appelée l'unité de la monade, et μ est la multiplication.

De plus, on demande que les diagrammes suivants commutent (associativité et neutre) :

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow{\mu} & T \end{array} \qquad \begin{array}{ccccc} Id_{\mathcal{C}}\ T & \xrightarrow{\eta T} & T^2 & \xleftarrow{T\eta} & T\ Id_{\mathcal{C}} \\ || & & \downarrow{\scriptstyle \mu} & & || \\ T & = & T & = & T \end{array}$$

Dans tout ce qui suit, on se place dans la catégorie $\mathcal{C} = \mathbf{Set}$ des ensembles.

Soit A un élément de **Set**, on dit que T envoie A dans la monade en TA. η_A est alors l'injection de A dans TA et représente le moyen effectif de plonger un objet de A dans la monade.

On peut voir l'unité de la monade comme la "porte d'entrée" dans la monade. Ensuite μ et son associativité permettent de faire des calculs dans la monade en garantissant que ce qui est fait est légitime. Pour voir plus précisément les implications de ce discours, voyons sans plus tarder l'utilisation de cette structure en Haskell.

2.3.2 Une monade en Haskell

2.3.2.1 Définition

En Haskell, une monade est un type de données qui définit un ensemble fixé de méthodes en respectant certaines relations (fixées elles aussi) entre ces méthodes. Les lois représentent les opérations η et μ ci-dessus (ou des équivalentes).

Ce schéma commun de définition et les relations qu'on impose entre les différentes fonctions (qui correspondent aux deux diagrammes commutatifs précédents) permettent de définir tout un ensemble de fonctions pour manipuler une instantiation particulière d'une monade.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Chacune de ces fonctions a une signification qui dépend de la monade, mais on trouve souvent des points communs entre les différentes monades et cela permet d'utiliser tout ceci avec une grande facilité.

Concrètement, une monade en Haskell est un moyen d'enrober une donnée avec des informations en plus qui lui sont attachées automatiquement, et de manière transparente. Ces informations supplémentaires circulent sans effort grâce aux lois de la monade.

Cela conduit par exemple à ne considérer non plus une simple valeur `Int` mais un objet de type `M Int` où `M` est une monade (ici `M` joue un rôle analogue à T ci-dessus). Alors on peut stocker des informations supplémentaires dans la monade, collées à la valeur `Int` de base. Et si on mène des calculs au sein de la monade, on fera les mêmes calculs que si on manipulait une simple donnée `Int`, mais cette fois-ci les informations collectées s'ajouteront au fur et à mesure du calcul donnant une plus grande flexibilité.

Ceci dit, on peut demander des comportements plus spécifiques à la monade pour que justement le comportement du calcul diffère du calcul normal, comme nous allons le voir.

À présent, essayons de comprendre tout cela à partir de code. Pour définir une monade, on définit un type, puis on en fait une instance de la *typeclass* `Monad` :

```
newtype MaMonade a = ...
instance Monad MaMonade where
  return :: a -> MaMonade a
  return a = ...

  (>>=) :: MaMonade a -> (a -> MaMonade b) -> MaMonade b
  m >>= k = ...
```

où les trois petits points sont l'implémentation effective de `MaMonade`.

La première ligne correspond à la définition du foncteur T : pour chaque a on définit Ta. La méthode `return` est le η qui permet de "rentrer" dans la monade. Enfin, `>>=` permet d'enchaîner les opérations sur la monade. On a déjà croisé cet opérateur dans notre cas particulier ci-dessus et nous allons voir qu'on a ici sa généralisation. Cet opérateur est aussi appelé `bind`.

Afin de comprendre le principe, voyons ce qui se passe sur un exemple. Supposons qu'on veuille envoyer un entier dans la monade, disons `2`. On va appeler `return 2`, qui est de type `MaMonade Int`. Cela veut dire qu'on a ici un objet de la monade qui contient comme valeur `2 : : Int`, en plus des données internes au fonctionnement de la monade, qui sont cachées de l'extérieur.

Maintenant supposons qu'on veuille appliquer une fonction qui transforme ce `Int` en sa représentation en `String`. On définit alors cette fonction qui agit au sein de la monade :

```
transform :: Int -> MaMonade String
transform x = return (Show x)          -- Show :: Int -> String fait ce qu'on veut
```

Notons qu'ici le terme `return` n'a pas le même sens qu'en `C` où il contrôle le flot d'exécution. Ici, c'est simplement le nom de l'injection dans la monade – qu'on aurait pu plus clairement appeler `inject`, mais c'est la convention.

À présent, si on se fie au typage, on est en droit d'appliquer :

```
-- ici, c'est : >>= :: MaMonade Int -> (Int -> MaMonade String) -> MaMonade String
return 2 >>=
transform
```

et cette expression a pour type `MaMonade String`. Le comportement attendu est le suivant : on injecte la valeur `2` dans la monade, puis on applique la fonction `transform` à la valeur `2`, tout en sachant que `>>=` fait en sorte que les deux opérations "s'enchaînent" bien en faisant ce qu'il faut au sein de la monade. Il faut éventuellement, selon la monade, modifier certaines des informations associées ou bien transmettre un état d'une ligne à l'autre.

Dans l'exemple du log qu'on a traité à la main, l'information à passer d'une ligne à l'autre était le texte que l'opération avait généré. Typiquement, c'est le genre de travail dont l'opérateur `bind` est censé s'occuper de manière transparente.

Attardons-nous quelques instants sur l'origine de l'appellation `bind`. On peut réécrire le code précédent comme ceci (pour qu'une fonction soit appelée en infixe en Haskell, il suffit de la mettre entre quotes) :

```
return 2 `bind` \x ->
transform x
```

et on peut alors lire le code comme ceci : exécuter `return 2`, extraire la valeur du résultat monadique, l'assigner ("bind") à `x` puis continuer l'évaluation, donc appeler `transform x`.

2.3.2.2 Fonctions usuelles dans une monade

Même sans connaître l'implémentation exacte d'une monade, on peut déjà construire des fonctions qui agiront dessus en accord avec ses deux lois. En effet, les deux méthodes `return` et `bind` définissent complètement le comportement de la monade entière.

C'est le cas des deux fonctions suivantes :

```
mapMaMonade       :: Monad MaMonade => (a -> b) -> (MaMonade a -> MaMonade b)
mapMaMonade f m   =  m `bind` (\a -> return (f a))

joinMaMonade      :: Monad MaMonade => MaMonade (MaMonade a) -> MaMonade a
joinMaMonade z    =  z `bind` (\m -> m)
```

Ces noms `map` et `join` viennent de la terminologie des listes. En effet, dans la monade des listes qu'on abordera ci-dessous, ces opérations correspondent aux opérations habituelles de `map` et de concaténation. Dans le cas général, on parle de "monadic map/join" pour insister sur le fait que la sémantique de ces opérations dépend complètement de la monade dans laquelle on les exécute.

On remarque que `joinMaMonade` qui vient d'être définie est l'équivalent de la transformation naturelle μ que l'on s'était donnée du côté catégorique. Il était donc bien légitime de définir la monade avec `return` et `bind` : c'est équivalent à la définition avec η et μ.

On peut montrer grâce aux relations de commutation que tout se déroule comme prévu : l'ordre d'évaluation dans `bind` ne change pas le résultat grâce à l'associativité et la neutralité de `return` garantit la correction des opérations. Par exemple, supposons que notre monade soit un interpréteur de lambda calcul, on a envie que :

```
interpretation (Add x (Add y z)) = interpretation (Add (Add x y) z)
```

Ce comportement peut ainsi être prouvé grâce aux lois de la monade [Wad01]. Si on ne remarquait pas qu'on est dans une monade et si on n'avait donc pas ces lois de commutation, on ne pourrait pas prouver que le comportement est le bon – sauf à faire un raisonnement ad hoc.

Il est aussi pratique de disposer d'un moyen d'exécuter une fonction pure au sein de la monade. Par exemple, on peut vouloir faire agir une fonction pure sur la valeur monadique sans changer du tout l'état de la monade. On définit donc la fonction :

```
liftMaMonade :: Monad MaMonade => (a -> b) -> MaMonade a -> MaMonade b
liftMaMonade f = \m -> m >>= (\a -> return (f a))
```

Au chapitre des fonctions essentielles d'un point de vue expressif, on notera aussi le `fold` qui se transforme aisément en *monadic fold*. Le `fold` permettant d'exprimer une classe très large de calculs [Hut99], il est intéressant de disposer d'une généralisation au cadre monadique. Cependant, je ne m'en servirai pas et je ne fais donc que le mentionner.

2.3.2.3 Relations à vérifier pour être une monade

Afin que la structure de données soit bel et bien une monade, nous avons dit que les relations d'associativité et de neutralité doivent être respectées (les deux diagrammes commutatifs de la vision catégorique doivent commuter).

Avec la définition des monades dans Haskell, les règles se réduisent à vérifier que :

```
(return a) `bind` k    =  k a                      -- Neutre à gauche
m `bind` return        =  m                        -- Neutre à droite
m `bind` (\a -> (k a) `bind` (\b -> h b)) =        -- Associativité
     (m `bind` \a -> (k a)) `bind` (\b -> h b)
```

Haskell n'a bien sûr pas de moyen automatique de garantir que ces règles sont bien respectées. C'est au programmeur de s'en assurer avec l'implémentation qu'il donne de `bind` et de `return`.

3 Différents usages des monades

Dans cette section, nous allons passons en revue quelques utilisations du concept de monades pour montrer que l'objectif qui était d'atteindre dans notre cadre purement fonctionnel des fonctionnalités impératives est bien rempli.

3.1 Court-circuiter grâce à la monade `Maybe`

3.1.1 La monade `Maybe`

La première utilisation des monades que l'on va aborder est très simple. Il s'agit d'un type qui représente la réussite ou l'échec d'une opération. On définit donc un type somme qui traite chacun des deux cas :

```
newtype Maybe a =
    Just a
  | Nothing
```

L'implémentation de la monade est alors triviale (on utilise le pattern matching pour savoir dans quel cas on se trouve dans `bind`) :

```
newtype MaMonade a = ...
instance Monad MaMonade where
  return :: a -> MaMonade a
  return a = Just a

  (>>=) :: MaMonade a -> (a -> MaMonade b) -> MaMonade b
  (>>=) (Just a) k = k a
  (>>=) Nothing k = Nothing
```

La définition de `return` ne pose pas de problème. Pour la définition de `>>=`, c'est simplement la syntaxe Haskell pour dire : si le calcul de ce qu'il y a à gauche du `bind` a échoué, alors le calcul global est échoué et on retourne directement échoué. Dans le cas contraire, on transmet l'argument à la fonction de calcul et on renvoie son résultat.

3.1.2 Exemple d'utilisation

Cette monade est très utile pour court-circuiter un calcul. Par exemple, disons qu'on recherche l'adresse mail d'une personne dans un annuaire. En code "imagé", on pourrait dire qu'on veut faire :

```
chercher la personne dans l'annuaire (et renvoyer la fiche annuaire)  >>=
chercher maintenant l'adresse e-mail dans la fiche annuaire
```

Si on se place dans la monade `Maybe` (je laisse les détails d'implémentation Haskell de côté), si la première recherche échoue, elle renverra `Nothing` et donc la deuxième recherche ne sera même pas exécutée puisque le pattern matching dans l'implémentation de `>>=` va complètement ignorer la deuxième expression. On utilise ici le caractère *lazy* de Haskell, c'est-à-dire qu'une expression n'est évaluée que lorsqu'on en a besoin. Typiquement, une expression passée en argument d'une fonction ne sera évaluée que lorsque l'argument est effectivement utilisé dans la fonction, pas lors de l'appel (convention Call By Name).

On a donc construit un analogue "propre" du `break` du `C++`.

3.2 Faire une boucle avec la monade des listes

3.2.1 La monade des listes

Elle se définit très simplement ainsi :

```
instance Monad [] where
  return :: a -> [a]
  return x = [x]

  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concat (map f xs)
```

`concat` est également appelée `flatten` dans certains langages. Elle a pour effet d'aplatir une liste de liste en une liste, en enlevant les imbrications internes. En réalité, on voit que le μ de la monade des listes est exactement `concat`.

3.2.2 Exemple d'utilisation

On peut faire une liste de tous les couples (i, j) pour $1 \leq i, j \leq 10$ comme ceci à l'aide de la monade :

```
[1..10] >>= \x ->
[1..10] >>= \y ->
            return (x,y)
```

Il est bien évident qu'à la place du `return` on pourrait faire l'opération qu'on veut, modulo la syntaxe Haskell.

On a donc bien imité une boucle en style fonctionnel pure, sans variable locale ! (En réalité, les variables sont bien sûr cachées sur la pile de récursion lors des appels de `map`.)

3.3 Imiter le style impératif grâce à la monade des états

La monade des états est plus sophistiquée que les deux précédentes. Afin de se concentrer sur le principe et non sur la syntaxe, on se permet d'écrire du code qui ne conviendrait pas directement. On pourra se reporter à [OGS08] pour plus de détails.

Cette fois-ci, on fait agir la monade (*i.e.* le foncteur T) sur les fonctions de type `S -> (a, S)`, où `S` est l'ensemble des états de la monade. Pour comprendre cette monade, il faut imaginer qu'un objet de la monade représente un *calcul*. Alors l'état `S` en cours (*i.e.* stocké dans l'objet monadique) représente l'état du calcul.

On lance le calcul sur une valeur initiale de l'état. Puis le connecteur `bind` transmet l'état tout au long du calcul, en prenant en compte le nouvel état modifié par l'application de toutes les étapes de calcul.

Dans l'exemple de la partie 2, l'état du calcul était simplement la liste des messages de log qui avaient été émis depuis le début du calcul. Dans un analyseur syntaxique par exemple, ce pourrait être le texte qu'il reste à parcourir.

On a donc le (pseudo-)code suivant :

```
-- Il faudrait faire un "newtype" mais cela compliquerait inutilement
type State s a = s -> (a, s)

instance Monad State where
  return :: a -> State s a
  return x = \s -> (x, s)          -- ne modifie pas l'état, renvoie juste la valeur

  (>>=) :: State a s -> (a -> State b s) -> State b s
  m >>= k =
      \s -> let (a, s') = m s
            in (k a) s'
```

La fonction de `bind` fait bien ce qu'il faut : l'argument `\s` reçu est l'état en cours au moment de l'appel. Il est transmis à la première opérande `m`, qui effectue son calcul en fonction de l'état. On note `s'` le nouvel état du calcul, et `a` le résultat de `m`. On fait agir `k` sur le résultat du calcul `a`, puis on le fait s'exécuter à partir du nouvel état `s'` qu'a laissé `m`.

Ainsi, après tout cela, on obtient bien une fonction de type `State s a` qui exécute `m` puis `k` en prenant en compte l'état du calcul lors de l'appel, et en le transmettant convenablement de `m` à `k`.

On dispose d'une monade (on peut le vérifier), et le comportement est bien celui auquel on s'attendait : on fait passer un état au cours d'un calcul de manière tout à fait pure. Mais chaque étape peut le modifier comme bon lui semble, créant ainsi un effet de bord et simulant l'utilisation de variable locale. En effet, on peut utiliser l'état comme variable locale (on l'a fait dans la section 2 sans le formuler ainsi).

Pour utiliser réellement la monade des états, on définit également deux fonctions `put` et `get` qui permettent respectivement de ne faire que changer l'état en cours, et de le récupérer. Leur implémentation est directe :

```
get :: State s s
get = \s -> (s, s)

put :: s -> State s ()
put snew = \s -> ((), snew)
```

On vérifie facilement avec la définition de `bind` que l'utilisation de ces deux fonctions fait bien ce à quoi on s'attend. Notons par ailleurs qu'il est bien entendu possible de prendre pour `a` un type plus compliqué présentant plusieurs champs (comme une `struct` en `C`). Et on fait alors passer "plusieurs variables" comme état de la monade, offrant toujours plus de flexibilité.

Dans la lignée de notre réflexion sur la pureté, l'intérêt philosophique de cette monade, c'est que l'effet de bord est très bien circonscrit. On sait exactement où il a lieu et comment. On sait même quelle fonction est susceptible d'en faire ou non d'après sa signature, parce qu'une fonction qui veut modifier l'état du calcul est contrainte à comporter `State a s` dans son type. Sinon, elle n'aura pas accès à l'état de la monade !

En conséquence, on se rend compte qu'on a un contrôle très précis sur le déroulement des opérations : on peut mixer du code pur (grâce à `lift` au lieu de `bind`) et du code presque pur en sachant toujours différencier les deux parties. C'est bien entendu très pratique pour corriger ou même prévenir les bugs, et pour garantir son code.

3.4 Interagir avec le monde grâce à la monade `IO`

Malgré toutes ces illustrations, on sent bien qu'il nous manque l'intéraction avec le monde extérieur. Au bout du compte, dans tous les exemples, les constantes sont fixées par le programmeur et donc le code ne fait qu'exécuter une suite prédéfinie d'opérations.

Évidemment, pour changer cet état de fait, il faut autoriser le programme à parler avec l'extérieur : le système de fichiers, le réseau, l'utilisateur, etc. Cela se fait par une monade qui est implémentée de manière standard en Haskell : la monade `IO`.

Pour comprendre son fonctionnement et son utilisation, je renvoie à [OGS08]. Le principe est que toute fonction qui veut intéragir avec l'extérieur doit passer par cette monade. Une telle fonction, qui est

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

par définition impure puisqu'elle peut retourner deux résultats différents si elle est appelée dans le même environnement (si le fichier qu'elle lit a changé par exemple), comporte `IO` dans son type. On peut ainsi très bien situer les endroits du code qui sont purs ou non, ce qui est l'objectif qu'on s'est fixé depuis le début.

Conclusion

Tout au long de cette étude, nous avons vu que malgré un paradigme fonctionnel qui peut paraître au départ paradoxal, la programmation fonctionnelle, par l'intermédiaire des monades, permet un contrôle très précis du déroulement d'un programme. On peut en effet très bien localiser les endroits où le code est pur, c'est-à-dire qu'il est "absolu" dans le sens où son exécution donnera toujours la même chose sur un même argument, et les endroits où l'exécution dépend de conditions extérieures au programme. C'est évidemment un atout majeur dans le développement d'un programme ou lors du débuggage : chaque partie est isolée de l'autre.

Les monades, qui sont des outils venus des catégories et dont la pertinence en programmation fonctionnelle a mis du temps à émerger, semblent s'emboîter à merveille dans ce cadre formel. Il faut tout de même souligner que leur utilisation en Haskell est soumise à la discrétion du programmeur car c'est à lui de vérifier que les lois de commutation de la structure de données qu'il définit sont bien vérifiées. Ce n'est qu'ensuite qu'il pourra considérer acquises les propriétés si agréables des monades que nous avons utilisées tout au long de notre discours.

Enfin, on peut signaler qu'il serait intéressant de savoir à quel point une monade peut modéliser des effets. Ici, nous avons modélisé notamment des changements d'état, mais on pourrait se demander si *tout* ce qu'on fait en programmation impérative peut être simulé par monade. C'est un des sujets actuels de recherche dans le domaine [Wad98].

Références

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Site de l'auteur*, 1999.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'REILLY, 2008.

[Wad92] Philip Wadler. Comprehending monads. *Site de l'auteur*, 1992.

[Wad97] Philip Wadler. How to declare an imperative. *Site de l'auteur*, 1997.

[Wad98] Philip Wadler. The marriage of effects and monads. *Site de l'auteur*, 1998.

[Wad01] Philip Wadler. Monads for functional programming. *Site de l'auteur*, 2001.