Principes de Programmation
Fiche de Révisions:
Foncteurs, Monades et Companie

22 juin 2017

1 Structures de donnés et structures de contrôle : exemple Liste et Maybe

La *liste chaînée* (ou simplement *liste*) est la plus simple et la plus utilisée des structures de donnés en programmation fonctionnelle. D'un autre côté, la "monade" *maybe* (anglais pour "peut-être") est la plus simple et la plus utilisée des structures de contrôle. Leur utilisation est donc très différentes, pourtant leur structure est similaire ; il apparaît en fait que les structures de données et de contrôle peuvent être généralement abstraite ensemble par un concept de foncteur, voire de monade.

Une structure de donnés est une structure qui contient des données... En programmation fonctionnelle, il s'agit généralement d'une représentation d'un ensemble de données arbitrairement grand et d'un même type ; de plus, ce type est généralement un paramètre de la structure, on peut donc mettre ce que l'on veut dedans tant qu'il s'agit toujours de la même chose. Une structure de donnés peut donc être divers (listes, tableau, arbres, graphe, table de hashage, etc...) et est associée à une algorithmique plus ou moins efficace (listes triés, arbres binaires de recherches, arbres rouge-noir, etc...).

Une structure de contrôle est un environement permettant d'utiliser un opérateur de contrôle (comme une boucle `for`, des exceptions, un break, des variables globales, des *threads* concurrents etc...). C'est une notion propre à la programmation fonctionnelle. [1] En effet, les programmes étant des donnés comme les autres, on peut les modifier avec d'autres programmes et donner sens à des opérateurs de contrôle qui n'aurait pas été prévus par le langage. Quelle utilité par rapport à des opérateurs implémentés nativement ? D'une part, les opérateurs dont on pourrait avoir besoin, ainsi que leur implémentation, évoluent. D'autre part, cela permet d'avoir un contrôle sur les opérateur autorisés et donc l'utilisation du code et le respect de ses invariants... [2]

1. Ce qui n'empêche pas d'en parler dans d'autres paradigmes.
2. En gros, cela permet d'empêcher un utilisateur de faire trop de conneries...

Les structures de donnés et de contrôles ont pas mal de comportements communs. D'une part, dans un cas comme dans l'autre, la structure est généralement un foncteur : on peut appliquer une fonction à chaque élément d'une structure de donnée et on peut toujours appliquer un programme qui n'utilise pas nos opérateurs de contrôle comme si de rien n'était. De plus, toutes les structures de contrôles et de nombreuses structures de données sont des monades : on peut les manipuler de l'intérieur, comme un environnement. Enfin, (quasiment) toutes les structures de données et certaines structures de contrôle sont des *foldable* : il est possible de les parcourir.

1.1 La liste, une structure de donnée typique

On rappel que les listes sont (moralement) implémentées ainsi :

```
data [] a =  a : [a]  |  []
```

Cela veut dire qu'une liste d'un certain type `a` est notée `[a]`, de plus, il s'agit soit d'une liste vide `[]`, soit de `x:l` où `x` est un élément de type `a` correspondant à la tête de la liste et `l ::[a]` est la suite de la liste.

C'est donc une structure de donnée contenant autant d'élément que l'on veut, on peut même écrire des listes infinies comme la liste suivante qui contient une infinité de `1` :

```
infinit1 :: [Int]
infinit1 = 1 : infinit1
```

La liste comme foncteur Une fonction extrêmement importante sur les listes est la fonction `fmap` (ou simplement `map`) qui va appliquer un programme à chaque élément de la liste, donnant une nouvelle liste mise à jours en tout points :

```
implement Functor [] where
fmap f []    = []
fmap f (x:l) = (f x) : (fmap f l)
```

Par exemples, si l'on considère la liste infinie `[0..]` des entiers de `0` à l'infinie, alors le programme :

```
nnplus1 :: [(Integer,Integer)]
nnplus1 = fmap (\n -> (n,n+1)) [0..]
```

est la liste[3] `[(0,1),(1,2],(2,3)..]` de tous les couples `(n,n1)`+ pour tout `n`.

Remarquez que l'on ne définit pas seulement le programme `fmap`, mais que l'on implémente la *class type* `functor`. Cela veut dire, tout d'abord, que le type de `fmap` n'est pas juste :

```
fmap :: (a -> b) -> [a] -> [b]
```

3. Attention cette notation n'est pas comprise par haskell.

mais qu'il est disponible pour tous les foncteurs :

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Cela est très utile si l'on veut plus tard utiliser une autre structure de donnée. De plus, implémenter la classe Functor indique (informellement) que l'on va bien appliquer la fonction à tous les membres de la liste.

L'implémentation d'un foncteur, et de `fmap`, est très importante pour une structure de donnée. Cela veut dire que la structure peut accueillir n'importe quel type de donnée et que son organisation ne dépend pas des données accueillies. En effet, on peut ainsi transformer toutes les données contenues d'un coup sans déranger la structures.

Remarque 1. En fait, certaines structures de données avancées ne sont pas réellement des foncteurs car elles supposent des choses sur les données. On en reparlera.

La liste comme foldable Une autre fonction extrêmement importante sur les listes est la fonction `foldl` qui va parcourir la liste et créer un résultat stocké dans un accumulateur :

```
implement Foldable [] where
foldl f acc []     = acc
foldl f acc (x:l) = foldl f (f acc x) l
```

Par exemple, si on considère `['a'..'z']`, la liste des caractères de `'a'` à `'z'`, alors le programme

```
vingtSix :: Int
vingtSix = foldl (\acc c -> (acc + 1)) 0 ['a'..'z']
```

va compter les lettres de l'alphabet.

Comme pour `fmap`, on ne définit pas seulement le programme `foldl`, mais que l'on implémente la *class type* `foldable`. Cela veut dire, tout d'abord, que le type de `foldl` n'est pas juste :

```
foldl :: (a -> b -> c) -> a -> [b] -> c
```

mais qu'il est disponible pour tous les *foldables* :

```
foldl :: (Foldable f) => (a -> b -> c) -> a -> f b -> c
```

Encore une fois, cela est très utile si l'on veut plus tard utiliser une autre structure de donnée. De plus, implémenter la classe `Foldable` indique (informellement) que l'on va bien traverser toute la structure.

Un `fold` permet de faire des recherches, mais aussi de récupérer la taille ou la composition de la structure. Par contre, l'utiliser n'est pas toujours la méthode la plus efficace.

1.2 Maybe, une structure de contrôle typique

On rappel que `Maybe` est définit comme suit :

```
data MaybeInt = Just Int | Nothing
```

Cela veut dire qu'une donnée de type `(Maybe a)` est soit un `(Just x)`, c'est à dire un élément de type `a` que l'on a mis dans une boite pour l'occasion, soit un `Nothing`, c'est à dire une sorte d'erreur. C'est ce que l'on utilise lorsque l'on fait un calcul alors qu'on n'est pas sure de l'existence d'un résultat.

Il s'agit d'une structure de contrôle au sens où on peut renvoyer une erreur basique `Nothing`, que l'on peut rattraper aisément avec un *pattern matching*. En fait, cela est très utile lorsque l'on a besoin d'une erreur comme "je n'ai pas trouvé ce qui était recherché", dont les circonstances sont connues et qui sera rattrapée très rapidement.

Maybe comme un foncteur On veut pouvoir utiliser un potentiel résultat comme s'il existait. Ainsi, une erreur est transmise, mais tout autre résultat peut continuer à être évalué. Cela veut dire que l'on peut appliquer une fonction à un `(Maybe a)` comme si c'était du `a`. Cela se fait aussi grâce à `fmap` :

```
implement Functor Maybe where
fmap f Nothing    = Nothing
fmap f (Just x)   = Just (f x)
```

Par exemples, si l'on considère le programme `lookup :: Int -> [a] -> Maybe a` qui (entre autre) récupère le $n$ième élément d'une liste s'il existe (et renvoi `Nothing` sinon), alors le programme :

```
lookupFoisDeux :: Int -> [Int] -> Maybe Int
lookupFoisDeux n = (map (*2)) . (lookup n)
```

va récupérer le $n$ième élément de la liste passée en second argument, puis va lu multiplier par 2 si elle l'a trouvé.

Pour une structure de contrôle, implémenter la classe Functor indique (informellement) que l'on va pouvoir travailler dans ce nouvel environnement comme si l'on était dans un environnement standard.

L'implémentation d'un foncteur, et de `fmap`, est très importante pour une structure de de contrôle. Cela veut dire que l'on est une réelle extension conservative de l'environnement normal : tant que l'on n'utilise pas d'opérateurs de contrôle, tout se passe comme dans l'environnement normal.

Maybe comme une monade Deux autre opérateurs essentiel de `Maybe` sont l'opérateur de retour et le *bind* `=<<` (et l'opération de retour associée). Le premier permet de dire "que l'on a maintenant le droit d'utiliser les opérateurs de contrôle", c'est essentielle pour importer une valeur calculée hors du `Maybe`. Le second dit que l'on peut composer deux programmes qui risques d'échouer. Ensemble, elles permettent vraiment de créer un nouvelle environnement où tout

est récupéré de l'ancien environnement en utilisant le *return* et où l'application est remplacé par le *bind* :

```
implement Monad Maybe where
return x          = Just x
f =<< Nothing     = Nothing
f =<< (Just x)    = f x
```

Par exemple, si l'on a une liste d'entier, que l'on veut récupérer le 3ième élément n puis rendre ne $n$ième élément, on précédera ainsi :

```
redirection :: [Int] -> Maybe Int
redirection l = (swap lookup l) =<< (lookup 3 l)
```

remarquons simplement que le `swap` permet d'inverser les deux arguments de `lookup` pour qu'elle prenne la liste avant l'entier.

Cet aspect "environnement" de la monade est mis en valeur avec la *do*-construction, qui est du sucre syntaxique pour l'utilisation de return et du *bind* dans une notation plus "impérative". En utilisant une *do*-construction, le programme `redirection` devient :

```
redirection :: [Int] -> Maybe Int
redirection l = do
  n   <- lookup 3 l
  m   <- lookup 3 n
  return m
```

1.3 Et inversement...

En fait, ces deux structures sont d'autant plus intéressante que les listes forment aussi une sorte de structure de contrôle et le maybe forme aussi une sorte de structure de donnée.

La liste comme monade La liste peut implémenter une monade avec : [4]

```
implement Monad [] where
return x          = [x]
f =<< []          = []
f =<< (x : l)     = (f x) ++ l
```

Par exemple, le programme :

```
multipleInt :: [Integer]
multipleInt = (\n -> [0..n]) =<< [0..]
```

va calculer la liste `[0,0,1,0,1,2,0,1,2,3,0,1,2,3,4...]`.

Intuitivement, la liste peut en effet être vu comme un environnement dégénéré. Il s'agit d'un environnement de "threads" qui sont calculés indépendemment. On peut aussi voir le résultat comme "les résultats possibles d'une évaluation non déterministe", ainsi il faut voir le programme précédent ainsi :

4. L'opérateur $++ :: [a] -> [a] -> [a]$ est la concaténation de listes.

– récupérons (non déterministiquement) un entier `n` de `[0,..]`,
– puis récupérons (non déterministiquement) un entier entre `0` et `n`.

Avec une *do*-construction, c'est exactement ce que l'on dit :

```
multipleInt' :: [Integer]
multipleInt' = do
  n <- [0..]
  m <- [0..n]
  return m
```

Maybe comme un foldable Le `Maybe` peut être vu comme une structure de donnés dégénérée avec zéro ou un élément. Ce n'est pas très intéressant, mais c'est suffisant pour implémenter la classe `Foldable` :

```
implement Foldable Maybe where
foldl _ acc Nothing    = acc
foldl f acc (Just x)   = f acc x
```

Entre structures de donnés et de contrôle (limites du programme) Il semble donc que les structures de données et les structures de contrôle ne soient pas si éloignées que cela. En fait, lorsque l'on imagine des objets suffisamment complexe, on se retrouve assez vite à franchir cette barrière. En effet, dès que l'on veut faire un peu de parallélisme, l'environnement créer doit pouvoir stocker et manipuler les différents threads. Inversement, dès que l'on veut une structure de donnée que l'on change dynamiquement (pensez à des requêtes *SQL* complexes), alors la structure de donnés devient une structure de contrôle dans laquelle on exécute du code.

Manipuler ce genre de structure complexe est extrêmement difficile, souvent beaucoup trop. C'est pour cela que l'on recherche un moyen d'abstraire les concept sous-jacent, ce qui nous permettrait de faire des manipulations d'apparences basiques (mais dont les implémentation peuvent être extrêmement complexe). Des concepts comme les foncteurs, les monades ou les foldable vont dans ce sens. Ceux-ci ont leur limites et ne peuvent pas encore traiter toutes les structures que l'on souhaiterais, mais elles permettent déjà une utilisation systématiques. À noter que cela n'est pas la fin de l'histoire, que les concepts importants vont évoluer, certains vont même disparaître, mais il semble néanmoins que l'on ai le bon niveau d'abstraction et que manipuler ces concepts d'aujourd'hui aident beaucoup à manipuler les concepts de demain.

2 Foncteurs

2.1 Informellement

Un foncteur est un type paramétré qui va contenir zéro, un ou plusieurs éléments de son paramètre sans les dénaturées. C'est donc un simple conteneur qui peut mettre des éléments ensemble, ou enrichir son contenu, mais pas le

changer fondamentalement. En particulier, si `Func` est un foncteur, on peut travailler sur `(Func a)` comme si s'était un élément de type `a` en utilisant juste des `fmap` aux bons endroits.

Il faut donc voir un foncteur comme une boîte qui contient l'objet réel. On n'a pas forcément de moyen d'ouvrir ou de fermer cette boite (même si c'est souvent le cas), par contre, si on a une fonction, on peut l'appliquer à ce qu'il y a dans la boîte.

2.2 Définition simplifiée et idéalisée

La classe `Functor` peut être vue comme la définition suivante :

```
class Foncteur f where
  fmap :: (a -> b) -> f a -> f b
-- requis: fmap
-- fmap id      = id
-- fmap (f . g) = (fmap f) . (fmap g)
```

Quelques remarques :

- un foncteur (ici désigné par la variable `f`) est un type paramétré. C'est pourquoi on peut écrire `(f a)` qui est le type obtenu en paramétrant `f` par `a`.
- Le type de `fmap` est équivalent au type suivent : [5]

  ```
  fmap :: (a -> b) -> (f a -> f b)
  ```

  On peut donc le voire comme transformant une fonction en une autre fonction. En fait, `fmap` transforme une fonction sur le contenu de type `a` en une fonction sur le contenant de type `(f a)`, mais en n'ouvrant pas la "boite" `f`.
- La première équation `(fmap id = id)` indique que l'on ne fait rien de plus que appliquer la fonctions aux contenus ; ainsi, si la fonction ne fait rien (c'est l'identité `id :: a->a`), la fonction résultante `(fmap id)` ne fait rien non plus (c'est l'identité `id :: f a -> f a`). Remarquez que l'on peut dérouler l'équation en :

  ```
  --  fmap (\x-> x) t  = t
  ```
- La seconde est moins intuitive, elle dit en gros que c'est bien cette fonction qui est appliquée, car même si on al décomposer en fonctions plus élémentaires `f.g`, on peut pareil décomposer `fmap (f.g)` en deux fonctions plus élémentaires `fmap f` et `fmap g`. Remarquez que l'on peut dérouler l'équation en :

  ```
  --  fmap (\x -> f (g x)) t = fmap f (fmap g t)
  ```
- (hors cours) D'un point de vue mathématique, ces équations sont très naturelles : elles disent que `fmap` conserve à la fois l'identité et la composition,

5. Souvenez-vous que $a->b->c$ signifie $a->(b->c)$ car on prend les deux arguments l'un après l'autre.

qui peuvent être vues comme des brique élémentaires de la programmation fonctionnelle. Ce dans une sorte d'extension du monoid appelée "catégorie" avec l'opérateur de composition et l'identité comme élément neutre.

2.3 Définition réelle

Pour une fois, la définition réelle est la même. L'objet est suffisamment simple et est compris depuis suffisamment longtemps pour ça.

3 Monades

3.1 Informellement

Une monade est un environement de travail, ou plus précisément une structure de contrôle. C'est un foncteur, donc une boîte autour d'une donnée, mais ce n'est pas ainsi qu'il est le plus aisé de le voire. Ce serait plutôt une boite vue de l'intérieur, c'est à dire un "environement" existant à côté du programme considéré. La particularité d'un tel environement, c'est que l'on peut considérer qu'il a toujours été là (juste qu'on ne le savait pas) et qu'il est unique. Ainsi, si on "découvre" deux fois l'existence d'un environement, ça reste le même, et si on le modifie à un moment, il reste modifié pour plus tard. Dans cette optique, le *return* n'est là que pour "informer de l'existence d'un environement" et le *bind* n'est là que pour remplacer l'application dans cet environement, la différence est qu'il peut "corréler les manipulations de l'environnement faites d'un côté et de l'autre".

Il faut noter que la notion de structure de contrôle étant très large, ce que l'on appel un environement a beaucoup de pouvoir ; il peut en particulier modifier la façon dont notre programme va s'exécuter si celui-ci lui en donne l'autorisation.

3.2 Définition simplifiée et idéalisée

La classe `Monad` peut être vue comme la définition suivante :

```
class Foncteur m => Monad m where
  return :: a -> m a
  (=<<)  :: (a-> m b) -> m a -> m b
-- requis: return, (=<<)
-- fmap f x           =   (return . f) =<< x
-- return =<< x       =   x
-- f =<< (return x)   =   f x
-- f =<< (g =<< x)    =   (\y. f =<< (g y) ) =<< x
```

Quelques remarques :

– Une monade est toujours un foncteur, mais ce n'est pas nécessaire d'implémenter `Functor` pour implémenter `Monad`, c'est fait automatiquement.

– En effet, la première équation permet d'écrire `fmap` à l'aide de `return` et de `(=<<)`. De plus, on vérifie en TD que les trois équations suivantes impliquent les règles du foncteurs, le `fmap` est donc bien toujours définit correctement.
– On peut aussi oublier l'héritage du foncteur et ne considérer que les trois dernières équations, puisque l'on peut toujours récupérer le foncteur plus tard. C'est ce qui est fait dans le cours.
– Une "fonction", dans notre environement qu'est la monade, est un objet de type `(a -> m b)` cela veut dire que l'on prend une valeur qui ignorerait l'existence de son environement et on va interagir avec un environement que l'on découvre seulement. Si la valeur d'entrée avait en fait déjà interagit avec l'environnement, ce n'est pas très grave, il faut seulement corréler les deux environement ; c'est pourquoi on utilise un bind.
– (hors cours) D'un point de vue mathématique, ces équations sont peu naturelles, mais elles sont équivalent à un autre ensemble de fonctions et d'équations la plus importantes des fonctions est la suivante :

  ```
  mmult :: m (m a) -> m a
  ```

  qui dit que si on traite avec un environement deux fois, alors c'est le même environement ; il faut juste "stacker" les modifications.

3.3 La *do*-construction

Bien souvent, lorsque l'on utilise une monade, on a tendance à se passer des opérateurs fonctionnels habituels. Il y a deux raisons à ça : La première est que la monade en elle même peut apporter une bonne partie e l'expressivité qu'apporteraient les opérateurs fonctionnels. a seconde est que la séquentialité est plus importante dans une monade ; l'ordre des interactions avec l'environnement est importante et se présente souvent plus aisément en programmation impérative.

C'est pourquoi il est possible (et souvent préférable) d'utiliser la *do*-construction, qui est une notation alternative pour les opérateurs de la monade en style très syntaxique.

Par exemple, dans la monade IO des entrées sorties (voire détail plus loin), on pour écrire le programme suivant :

```
nom :: IO String
nom = do
  putStrLn ``Comment vous appelez-vous?''
  nom <- getLine
  putStrLn (``Bienvenu'' ++ nom)
  return mot
```

Il y a principalement trois instructions utilisés dans une do-construction :
– Les "effets" : ce sont des commandes de type `(m ())`, c'est à dire la monade autour du type `()` qui ne dispose que d'une valeur [6], elle même notée

6. C'est la même chose que Void en Java ou unit en Ocaml

`()`. Ainsi un effet va interagir avec l'environnement, sans "renvoyer de résultat". Par exemple, la fonction `putStrLn :: String -> IO ()` va créer un effet une fois que l'on lui donne une string.
- Les "assignements" : Ce sont des commandes de la forme `x <- fun` où `x` est une variable et `fun` est une fonction de type `(m a)`. Il s'agit de "récupérer" un élément `x::a` contenu dans la monade. On peut utiliser cette variable par la suite. Ainsi, `(nom <- getline)` va récupérer un mot écrit par l'utilisateur à l'aide de la fonction `getline:: IO String`, on va ensuite mettre le résultat dans la variable `nom`.
- Le "résultat" : à la fin, on veut généralement retourner une valeur pour l'ensemble de la *do*-structure ; on utilise alors souvent `return val` où `val :: a` est le contenu retourné de la monade. À noter que l'on peut en fait écrire n'importe quel valeur `mVal :: m a`. on n'est pas non obligé de renvoyer quoi que ce soit à la fin, dans ce cas on rend le type `m ()`.
- (limite du programme) On peut aussi avoir d'autres structures, en particulier un `if`, mais attention le `then` et le `else` du `if` sont accompagnés d'un nouveau `do`.

Attention, ceci n'est que du "sucre syntaxique", cela veut dire que c'est simplement une notation qui est encodée en interne comme une suite de *bind*. Affin de comprendre cet encodage, remarquons d'abord qu'un effet `eff::m()` pourrait aussi être effectué en écrivant l'assignement `x<-eff` qui récupère une valeur `x::()` qui ne sera pas utilisée. Ainsi, on peut considérer que la construction est de la forme :

```
do
  x1 <- fun1
  x2 <- fun2
  ...
  xn <- funn
  final
```

ce qui s'encode par :

```
  fun1 >>= {\x1 ->
    fun2 >>= {\x2 ->
      ...
        funn >>= (\xn ->
          final
          )
      ...
      )
    )
```

3.4 Quelques fonctions utiles (hors programme)

```
liftM2 :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f x y = do
```

```
  x' <- x
  y' <- y
  return (f x y)
```

qui est une sorte de map pour une fonction à deux arguments. Il y a aussi les versions `liftM3`, `liftM4` et `lift5` pour les fonctions à 3, 4 ou 5 arguments.

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

qui est la composition pour des fonctions monadiques.

```
when :: Monad m => Bool -> m () -> m ()
```

Particulièrement utile dans une *do*-construction : on applique l'effet du second argument seulement si le premier est vrais.

3.5 Définition réelle (hors programme)

La classe des monades de Haskell est en fait un peu plus complexe, mais connaître ces détails n'est pas vraiment utile pour travailler avec :

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a-> m b) -> m b
   (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
-- requis: return, (>>=)
--  pure                    = return
--  f <*> x                 = f >>= (\g. x >>= (return . g))
--  m >> k                  =  m >>= \_ -> k
--  return a >>= k          =  k a
--  m >>= return            =  m
--  m >>= (\x -> k x >>= h) =  (m >>= k) >>= h
```

Quelques remarques :

- la classe n'hérite pas de `Functor` mais de `applicative` qui elle même hérite de `Functor`. Ça ne fait pas beaucoup de différence dans les fait...
- On ne définit pas `(=<<)` mais seulement `(>>=)` qui est son inverse (prend l'argument avant la fonction). Ça ne fait pas beaucoup de différence et n'est pas très naturel. Avec de la pratique le sens `(>>=)` peut néanmoins avoir ses avantages car il respecte mieux la séquentialité...
- On a deux nouveaux opérateurs : `(>>)` et `fail`. Ceux ci sont rarement utile, mais ils sont là car ils peuvent avoir des optimisations optimisés. La première sert à oublier le contenu d'une monade tout en conservant la structure ; le second indique ce qu'il se passe lorsqu'une exception machine intervient dans une monade.
- Les deux premières équations sont les définitions des opérateurs de `aplicative`.

4 Foldables (limite du programme)

5 Autres foncteurs et monades connus

5.1 Les couples

De la même façon que le Maybe est une sorte de liste avec 0 ou 1 élément, les couples forment des listes avec exactement 2 éléments. Il n'est donc pas surprenant d'avoir un foncteur, ou même un foldable :

```
data Couple a = Couple a a
implement Functor Couple r  where
  fmap f (x,y)  = (f x, f y)

implement Foldable Couple where
  foldl f acc (x,y) = f (f acc x) y
```

(Hors programme) Par contre, ce n'est pas une monade au même sens que la liste. Concrètement, c'est parce que le *retrun* utilise la possibilité d'avoir un unique élément, et le bind va multiplier le nombre d'éléments (ce qui est suffisant lorsqu'il y a 0 ou 1 élément, mas pas 2). Intuitivement, Si on n'a le droit que à deux threads, il faut prendre une décision lorsque l'un de ces threads essai de créer de nouveaux threads... Il faut alors oublier un thread existant. L'implémentation suivante est viable, mais elle n'est ni canonique, ni utile, ni utilisée (elle n'apparaît d'ailleurs pas dans la bibliothèque standard)

```
data Couple a = Couple a a
implement Monad Couple r  where
  return x    = (x,x)
  f =<< (x,y) = let (x',_) = f x in let (_,y') = f y in (x',y')
```

5.2 Les valeurs couplées (limite du programme) : (,) r

La notation est subtile, mais le concept très simple. Le type paramétré `((,) r)` va associer à chaque `a` le type des couples `(r,a)`. On considère donc les couples formés d'un élément de `r` et d'un élément du type paramètre `a`. Ceci forme un foncteur :

```
implement Functor (,) r  where
  fmap f (v,x)  = (v, f x)
```

Dans un tel couple, il y a qu'une valeur du type paramètre, on ne modifie donc que celle-ci... C'est pareil pour le foldable, on ignore simplement la valeur de gauche :

```
implement Foldable (,) r where
  foldl f acc (v,x) = (f acc x)
```

(Hors programme) Si on veut en faire une monade par contre, il faut supposer en plus que `r` est un `Monoid`. Intuitivement, on peut "écrire" des choses dans la valeur de gauche (de type `r`). C'est pourquoi on l'appel aussi la monade d'écriture, ou *writer* :

```
implement (Monoid r) => Monad (,) r  where
  return x    = (mempty,x)
  f =<< (v,x) = let (w,y) = f x in (v<>w,y)
```

5.3 Image de la flèche : (->) r

Encore une notation est subtile pour un concept très simple. Le type paramétré `((->) r)` va associer à chaque `a` le type fonctionnel `(r -> a)`. On considère donc les fonctions de `r` dans le type paramètre `a`. Ceci forme un foncteur :

```
implement Functor (->) r  where
  fmap f g    = f . g
```

L'idée c'est que l'on a un contenter abstrait qui à chaque élément de `r` associe un élément de `a`. Si on veut modifier tous les éléments de `a`, il suffit de composer les fonctions.

En fait on a même une monade :

```
implement Monad (->) r  where
  return x   = \v -> x
  f =<< g    = \v -> f (g v) v
```

L'idée est de considérer la variable `v::r` comme une variable global accessible uniquement en lecture. Lorsque l'on est dans la monade `r->a`, on calcule du `a` tout en ayant accès à une même variable `v::r`. Cette monade est aussi appelée monade de lecture, ou *reader*.

(hors programme) Si on peut parcourir `r`, c'est à dire si il est borné et énumerable, alors on a même un foldable :

```
implement (Bounded r,Enum r) => Foldable (->) r  where
  foldl f acc g = foldl f acc (fmap g [minBound,maxBound])
```

Cela vient de la même intuition que pour le foncteur : pour chaque élément de `r` on a une valeur de `a`...

5.4 Yiell (limite du programme)

Il s'agit d'une structure pas très standard vue en TD. Cet exemple est néanmoins intéressant car simple et mettant en valeur une des possibilités de contrôle offerte par la monade. On définit donc :

5.5 Les exceptions : Except

5.6 Les entrées/sorties : IO

5.7 Les états : State

5.8 Les continuation : Cont (limite du programme)