Programmation – TD11 : Monades

{ Emmanuel.Jeandel, Jeremie.Detrey } @ens-lyon.fr

5/6 janvier 2005

Pour votre plus grand plaisir à tou-te-s, on finit par un TD d'OCaml !

1 Un brin de théorie

1.1 Introduction

On va dans cette partie donner une définition alternative des monades, et montrer l'équivalence entre les deux définitions. Pour ceux qui n'ont pas envie de faire trop de théorie (même simple), faites uniquement les questions **2** et **4**.

En cours, on a défini une monade comme la donnée :
- d'un constructeur de type `'a m`;
- d'une fonction return : `'a -> 'a m`;
- d'une fonction »= : `'a m -> ('a -> 'b m) -> 'b m` qu'on notera de façon infixe ;

vérifiant les trois axiomes :

1. $(\text{return}\, a) \gg\!= k \equiv (ka)$;
2. $n \gg\!= \text{return} \equiv n$;
3. $m \gg\!= (\lambda x.\, ((kx) \gg\!= c)) \equiv (m \gg\!= k) \gg\!= c$.

En vrai, en théorie des catégories[1], une monade est la donnée :
- d'un constructeur de type `'a m`;
- d'une fonction lift : `('a -> 'b) -> ('a m -> 'b m);`
- d'une fonction return : `'a -> 'a m;`
- d'une fonction join : `('a m) m -> 'a m;`

vérifiant les six axiomes suivants :

a. $\text{lift}\, \lambda x.f(gx) \equiv \lambda x.\, \text{lift}\, f\ (\text{lift}\, g\ x)$;
b. $\text{lift}\, f\ (\text{join}\, x) \equiv \text{join}(\text{lift}\ (\text{lift}\, f)\ x)$;
c. $\text{lift}\, f\ (\text{return}\, x) \equiv \text{return}(fx)$;
d. $\text{join}(\text{lift}\, \text{join}\, x) \equiv \text{join}(\text{join}\, x)$;
e. $\text{join}(\text{return}\, x) \equiv x$;
f. $\text{join}(\text{lift}\, \text{return}\, x) \equiv x$.

[1] Qu'on va traduire ici d'une façon bâtarde.

1.2 Questions

On se donne un monoide quelconque `M` de neutre `e` et d'opération notée `#` et on considère la monade donnée par :

```
type 'a m = (M * 'a)
let lift f (x,y) = (x,(f y))
let return x = (e,x)
let join (x,(y,z)) = ((x#y),z)
```

1. Vérifiez que les axiomes d à f sont bien satisfaits et déduisez-en une interprétation possible de ces axiomes.

On se donne maintenant une monade définie au premier sens du terme.

2. Définissez les termes join et lift.
3. Vérifiez que les axiomes a à f sont bien satisfaits avec votre définition.

On se donne maintenant une monade définie au deuxième sens du terme.

4. Définissez »=.
5. Vérifiez que les axiomes 1 à 3 sont satisfaits (ne passez pas trop de temps sur 3).

2 Des monades intéressantes

2.1 Préambule

On suppose avoir défini une monade.

1. Quel est le résultat du code suivant ?

```
return 3                >>=
fun x -> return (x * x) >>=
fun y -> return (y + 2) >>=
fun z -> return (z + 3)
```

En Haskell, le même code peut aussi s'écrire de la façon plus jolie suivante

```
do
  x <- return 3
  y <- return x * x
  z <- return y + y
  return z + 3
```

D'un coup, on comprend mieux ce que ça fait...

2.2 Monade d'erreur

On définit la monade d'erreur par :

```
type 'a m = Valid of 'a | Error

let return x = Valid x
let (>>=) x f = match x with
  | Valid y -> (f y)
  | Error -> Error

let fail () = Error
```

1. Si ça vous amuse, vérifiez que c'est une monade.
2. Réécrivez ce petit bout de code et testez-le sur les deux exemples suivants :

```
return 3                >>=
fun x -> return (x * x) >>=
fun y -> return (y + 2) >>=
fun z -> return (z + 3)
```

```
return 3                       >>=
fun x -> return (x * x)        >>=
fun y -> if y = 9 then fail()
         else return (y + 2)   >>=
fun z -> return (z + 3)
```

2.3 Monade des parties, à vous de jouer

On définit formellement la monade des parties par :
- `'a m` correspond au type des ensembles à valeurs dans `'a` ;
- return est l'application qui à x associe $\{x\}$;
- »= est l'application qui à X et à f associe $\cup_{x\in X} f(x)$.

1. Si ça vous amuse, vérifiez que c'est une monade.
2. À quoi une telle monade peut-elle servir, intuitivement ?
3. Programmez cette monade.
   Remarque : pour simplifier, on va poser `type 'a m = 'a list` et faire des concaténations de listes au lieu d'unions. Ça ne donnera donc pas exactement le même résultat, mais ça en sera suffisament proche pour que ça ne soit pas gênant.
4. Testez votre monade sur l'exemple :

```
return 3                >>=
fun x -> return (x * x) >>=
fun y -> return (y + 2) >>=
fun z -> return (z + 3)
```

5. Ajoutez une fonction `(||) : 'a m -> 'a m -> 'a m` de sorte que le code suivant :

```
return 13                                >>=
fun x -> return (x * 4)                  >>=
fun w -> (return (w*4) || return (w-1)) >>=
fun y -> (return y || return (y*8))
```

affiche `[208; 1664; 51; 408]`.

6. Ajoutez une fonction `test : bool -> unit m` qui permet de choisir des branches non déterministes. Ainsi, le code suivant affichera uniquement `1664` :

```
return 13                                >>=
fun x -> return (x * 4)                  >>=
fun w -> (return (w*4) || return (w-1)) >>=
fun y -> test (y > 60)                   >>=
fun _ -> return (y * 8)
```

Indication : on pourra commencer par définir les fonctions `fail : unit -> unit m` et `succeed : unit -> unit m` tel que `succeed` soit plus ou moins une instruction vide de contenu, et où `fail` provoque la suppression de la branche non déterministe choisie. De sorte qu'on pourra ensuite écrire :

```
let test x = if x then succeed () else fail ()
```

2.4 Variables

On veut maintenant faire une monade qui permettent de faire des refs (sur des entiers). On va représenter un environnement par une liste d'association :

```
type env = (string * int) list
type 'a m = env -> 'a * env
```

1. Écrivez return et »=.
2. Écrivez les fonctions `set : string -> int -> unit m` et `get : string -> int m`.
3. Testez votre code sur l'exemple suivant :

```
(return 6                 >>=
 fun x -> set "a" 4       >>=
 fun _ -> return (x * 8)  >>=
 fun w -> get "a"         >>=
 fun y -> return (y + w)  >>=
 fun z -> return (z * 32)     ) []
```

que l'on peut aussi lire de la façon suivante :

```
x <-  return 6
_ <-  set "a" 4
w <-  return x * 8
y <-  get "a"
z <-  return y + w
return z * 32
```