

Cours 6 : Structures monadiques

2020 - 2021

- monades simples, additives, généralisées
- applications: monade(s) non-déterministe(s), d'état, d'état linéaire

Introduction

- considérons le calcul des permutations
- le type `'a list list` rend le programme peu clair
- abstraction des ens. de permutations difficile
- quid des approches itératives, aléatoires, probabilistes, ... ?
- la structuration monadique est une solution

```
(* insertions : 'a -> 'a list -> 'a list list *)
```

```
let rec insertions e l =
```

```
  match l with
```

```
  | [] -> [[e]] (* insertion de e en fin *)
```

```
  | t::q -> (e::l) (* insertion de e avant t *)
```

```
  ::  
    List.map (fun l -> t::l) (* on ajoute t en tete des ... *)
```

```
    (insertions e q) (* insertions de e apres t, i.e. dans q *)
```

```
(* permutations : 'a list -> 'a list list *)
```

```
let rec permutations l =
```

```
  match l with
```

```
  | [] -> [[]]
```

```
  | e::q -> List.flatten (List.map (fun sigma -> insertions e sigma) (permutations q))
```

Interface

```
module type FONCTEUR =  
sig  
  type 'a t  
  val map: ('a -> 'b) -> ('a t -> 'b t)  
end
```

Propriétés

$\text{map id} = \text{id}$
 $\text{map } (f \circ g) = (\text{map } f) \circ (\text{map } g)$

Interface

```
module type MONADE =  
sig  
  include FONCTEUR  
  val return : 'a -> 'a t  
  val bind   : ('a -> 'b t) -> ('a t -> 'b t)  
end
```

Propriétés

$\text{map } f = \text{bind } (\text{return} \circ f)$

$\text{bind return} = \text{id}$

$(\text{bind } f) \circ \text{return} = f$

$(\text{bind } f) \circ (\text{bind } g) = \text{bind } ((\text{bind } f) \circ g)$

$\text{bind } f [a1, a2, \dots, an] =$
or the list: $(f a1) @ (f a2) @ \dots @ (f an)$

Interface alternative

```
module type MONADE =  
sig  
  include FONCTEUR  
  val return : 'a -> 'a t  
  val (>>=) : 'a t -> ('a -> 'b t) -> 'b t      [a1, a2, ... ,an] >>= f  
end
```

Propriétés alternatives

```
map f x = (x >>= (fun v -> return (f v)))  
(x >>= return) = x  
(return v >>= f) = f v  
((x >>= (fun v -> f v)) >>= g) = (x >>= (fun v -> f v >>= g))  
= (x >>= fun v -> f v >>= g)
```

Interface alternative - 2

```
module type MONADE =  
sig  
  include FONCTEUR  
  val return : 'a -> 'a t  
  val (>=>) : ('c -> 'a t) -> ('a -> 'b t) -> 'c -> 'b t  
end
```

$(a \rightarrow b \ t) \ (b \rightarrow c \ t)$
 $f \gg g : a \rightarrow ct$

Propriétés alternatives - 2

```
map f = (id >=> (fun v -> return (f v)))  
(f >=> return) = f  
(return >=> g) = g  
((f >= g) >=> h) = (f >=> (g >=> h))  
= (f >=> g >=> h)
```

Quelques monades simples

- monade “identité”:

```
module Id : MONADE =  
struct  
  type 'a t = 'a  
  let map f x = f x  
  let return x = x  
  let (>>=) x f = f x  
end
```

- monade “option” / “exception” / “erreur”:

```
module Option : MONADE =  
struct  
  type 'a t = 'a option  
  let map f x = match x with  
    | None -> None  
    | Some v -> Some (f v)  
  let return x = Some x  
  let (>>=) x f = match x with  
    | None -> None  
    | Some v -> f v  
end
```


Quelques monades simples

- monade “liste”:

```
module Liste : MONADE =  
struct  
  type 'a t = 'a list  
  let map f l = List.map f l  
  let return x = [x]  
  let rec (>>=) l f =  
    match l with  
    | [] -> []  
    | t::q -> f t @ (q >>= f)  
    (* ou encore *)  
  let bind f x = List.flatten (List.map f x)  
end
```

- question à 0.02€: quelle différence avec la monade “option” ?
- applicable aux permutations ?

Quelques monades simples

- monade “ensemble” (listes sans doublons)
- cette monade est en fait appelée la monade “non-déterministe” **NDET**

```
module NDET : MONADE =  
struct  
  type 'a t = 'a list  
  let add l x = if List.mem x l then l else x :: l  
  let union l1 l2 = List.fold_left add l1 l2  
  let map f l = List.fold_left (fun x -> add (f x)) [] l  
  let return x = [x]  
  let rec (>>=) x f =  
    match x with  
    | [] -> []  
    | t :: q -> union (f t) (q >>= f)  
end
```

- toutes ces monades simples sont finalement ensemblistes
- ensembles à $\{ 1, 0 \text{ ou } 1, n, n \text{ sans doublons} \}$ éléments

- `permutations : 'a list -> 'a list NDET.t`
- `insertions : 'a -> 'a list -> 'a list NDET.t`
- version monadique de `permutations`:

```
let rec permutations l =  
  match l with  
  | [] -> NDET.(return [])  
  | e::q -> NDET.(permutations q >>= fun perm_q -> insertions e perm_q)
```

- on ne peut (toujours) pas définir `insertions` !

Approche monadique: monades additives

Interface

```
module type MONADE_PLUS =  
sig  
  include MONADE  
  val zero : 'a t  
  val (++) : 'a t -> 'a t -> 'a t  
end
```

Propriétés

$\text{zero} ++ a = a ++ \text{zero} = a$

$(a ++ b) ++ c = a ++ (b ++ c)$

$\text{zero} >>= f = \text{zero}$

$(a ++ b) >>= f = (a >>= f) ++ (b >>= f)$

$x >>= (\text{fun } _ \rightarrow \text{zero}) = \text{zero}$

$x >>= (\text{fun } v \rightarrow f\ v ++ g\ v) = (x >>= f) ++ (x >>= g)$

Monades additives

- les monades `Id` et `Option` ne sont **pas** additives (pas de `++`)
- `Liste` n'est **pas** additive (pas linéaire, à l'ordre près)
- par exemple¹: `Liste .((r 1 ++ r 2) >>= fun v -> r v ++ r (v+2))` \neq
`Liste .(((r 1 ++ r 2) >>= fun v -> r v) ++ ((r 1 ++ r 2) >>= fun v -> r (v+2)))`
- la monade `NDET` est additive:

```
module NDET : MONADE_PLUS =  
struct  
  :  
  :  
  let zero = []  
  
  let (++) = union  
end
```

¹`return` est abrégé en `r`

- on peut maintenant définir `insertions` : `'a -> 'a list -> 'a list NDET.t` :

```
let rec insertions e l =  
  match l with  
  | [] -> NDET.(return [e])  
  | t :: q -> NDET.(return (e::l)  
    ++  
    map (fun l -> t::l)  
      (insertions e q))
```

- en fait, l'implantation de `NDET` est libre (dans le respect des lois monadiques), donc gain en modularité très important
- on étudiera quelques variations autour de `NDET`
- la plupart des monades possèdent des opérations propres, en fonction de ce qu'elles représentent. On pourrait par exemple ajouter dans `NDET`:
`filter` : `('a -> bool) -> 'a t -> 'a t`

- une monade M sert à encapsuler/représenter un “effet”, par dessus un calcul normal
- le type $'a\ M.t$ est une valeur de type $'a$, obtenue au moyen de l'effet $M.t$
- le type $'a\ M.t$ est le plus souvent abstrait, donc l'usage de la monade M est contaminant
- pour isoler l'usage de M , on définit une fonction $run : 'a\ M.t \rightarrow 'a\ result$, où $'a\ result$ dépend de $'a$, mais pas de $M.t$
- la composition de monade, i.e. d'effets, est complexe, il faut *a minima* construire une bijection $'a\ M1.t\ M2.t \leftrightarrow 'a\ M2.t\ M1.t$

Variations sur NDET

- il existe d'autres implantations “ensemblistes” possibles de NDET
- par exemple, sans structure de données (avec consommateur):

```
module NDET : MONADE_PLUS =  
struct  
  type 'a t = ('a -> unit) -> unit  
  
  let return v = fun k -> k v  
  let map f s = fun k -> s (fun a -> k (f a))  
  let (>>=) x f = fun k -> x (fun v -> f v k)  
  let zero = fun k -> ()  
  let (++) a b = fun k -> (a k; b k)  
end
```

- `k` est le consommateur/client
- implantation de type “Push”

Variations sur NDET

- ou encore, avec tirage aléatoire (en distinguant l'ens. vide):

```
module NDET : MONADE_PLUS =  
struct  
  type 'a t = (unit -> 'a) option  
  let return v = Some (fun () -> v)  
  let map f s =  
    match s with  
    | None -> None  
    | Some g -> Some (fun () -> f (g ()))  
  let (>>=) x f =  
    match x with  
    | None -> None  
    | Some g -> f (g ())  
  let zero = None  
  let (++) a b =  
    match a, b with  
    | None , - - - - -> b  
    | - - - , None -> a  
    | Some f, Some g -> Some (fun () -> if Random.bool () then f () else g ())  
end
```

- monade pseudo-additive, on a en général $(a ++ b) \neq (a ++ b) !$

Application à la recherche de solutions

- on cherche un élément d'un type 'a qui satisfait une propriété
critere : 'a -> bool
- on procède de proche en proche, en visitant différentes positions
- en utilisant une fonction voisinage : 'a -> 'a list qui donne les voisins d'un élément donné
- applications: rendre la monnaie, le compte est bon, labyrinthe, ...

```
(* recherche_liste : ('a -> 'b list) -> 'a list -> 'b list *)
```

```
let rec recherche_liste cherche positions =
```

```
  match positions with
```

```
  | [] -> [] (* ECHEC ! *)
```

```
  | position :: queue -> match cherche position with
```

```
    | [] -> recherche_liste recherche queue
```

```
    | solution -> solution;; (* OK ! *)
```

```
(* recherche : ('a -> 'a list) -> ('a -> bool) -> 'a -> 'a list *)
```

```
let rec recherche voisinage critere position =
```

```
  if critere position then [ position ] (* OK ! *) else
```

```
  match recherche_liste (recherche voisinage critere) (voisinage position) with
```

```
  | [] -> [] (* ECHEC ! *)
```

```
  | solution -> position :: solution ;;
```

Application de l'application: rendre la monnaie

- le type **position** représente l'état courant du problème: la monnaie à rendre et l'ensemble des pièces disponibles dans la caisse
- le **critere** (solution) est que le montant à rendre soit nul
- les actions possibles au **voisinage** d'une position donnée sont: soit rendre la première pièce disponible; soit "jeter" cette première pièce
- pieces_rendues** retrouve les pièces rendues à partir du chemin solution

```
type position = int * int list
let critere (a_rendre, caisse) = a_rendre = 0;;
let voisinage (a_rendre, caisse) =
match caisse with
| [] -> []
| p::q -> (if p <= a_rendre then [(a_rendre-p, q)] else []) @ [(a_rendre, q)];;
let rec pieces_rendues pos_list =
match pos_list with
| [] -> failwith " impossible "
| (_, _)::[] -> []
| (n1, _)::(n2, c2)::q -> (n1-n2)::pieces_rendues ((n2, c2)::q);;
let rendre_monnaie a_rendre caisse =
pieces_rendues (recherche_voisinage critere (a_rendre, caisse));;
```

Application à la recherche de solutions

1. utiliser le type 'a NDET.t à la place de 'a list

```
recherche_liste : ('a -> 'b NDET.t) -> 'a NDET.t -> 'b NDET.t  
recherche : ('a -> 'a NDET.t) -> ('a -> bool) -> 'a -> 'a list NDET.t
```

2. utiliser zero et ++:

```
let rec recherche_liste cherche positions =  
  match positions with  
  | [] -> zero  
  | position :: queue -> cherche position ++ recherche_liste cherche queue
```

3. reconnaître bind:

```
let recherche_liste cherche positions = positions >>= cherche
```

4. le traitement du résultat de recherche_liste est également un bind:

```
let rec recherche voisinage critere position =  
  if critere position then return [position] else  
  voisinage position >>=  
  recherche voisinage critere >>=  
  fun chemin -> return (position::chemin)
```

La monade **READER**

- permet d'accéder, de façon transparente, à une donnée non modifiable (habituellement un environnement), qui est transportée implicitement dans tout le programme
- le type $('a, 'c) \text{ t}$ est fonctoriel en $'a$, pour $'c$ constant

```
module READER =  
struct  
  type  $('a, 'c) \text{ t} = 'c \rightarrow 'a$   
  let return  $(v : 'a) : ('a, 'c) \text{ t} = \text{fun } _ \rightarrow v$   
  let  $(>>=) (x : ('a, 'c) \text{ t}) (f : 'a \rightarrow ('b, 'c) \text{ t}) : ('b, 'c) \text{ t} = \text{fun } c \rightarrow f(x\ c)\ c$   
  let map  $(f : 'a \rightarrow 'b) (x : ('a, 'c) \text{ t}) : ('b, 'c) \text{ t} = \text{fun } c \rightarrow f(x\ c)$   
  let get  $: ('c, 'c) \text{ t} = \text{fun } c \rightarrow c$   
  let run  $(x : ('a, 'c) \text{ t}) (c : 'c) : 'a = x\ c$   
end
```

Application de READER à l'évaluation d'expressions

- évaluation d'expressions arithmétiques (avec variables):

```
type expr = | Const of int | Var of string | Add of expr * expr
let rec eval env e =
  match e with
  | Const c      -> c
  | Var v        -> List.assoc env v
  | Add (e1, e2) -> eval env e1 + eval env e2
```

- application de la monade READER:

```
type expr = | Const of int | Var of string | Add of expr * expr
let eval env expr =
  READER.(
    let rec eval e =
      match e with
      | Const c      -> return c
      | Var v        -> return (List.assoc get v)
      | Add (e1, e2) -> eval e1 >>= fun v1 -> eval e2 >>= fun v2 -> return (v1 + v2)
    in run (eval e) env)
```

- l'addition est une instance de `map2`: `('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t`
- on a fait disparaître l'environnement `env` de la fonction récursive `eval`

- variable d'état (modifiable), entrées-sorties, exceptions, non-déterminisme, calcul probabiliste, calcul quantique, environnement, journalisation, transactions, continuations, . . .
- utiles même dans un langage où les effets sont nativement présents comme OCAML
- utiles également en logique (double négation), en topologie (fermeture), etc
- d'autres monades seront couvertes en TD et TP