TD7 : Structures monadiques

Thèmes et objectifs

— Monades simples, additives, généralisées

▷ **Support étudiant**

1 Approche monadique

Pour aborder les monades, on reprend l'exemple du calcul des permutations d'un ensemble.

```
(* 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))
```

On s'aperçoit qu'on écrit souvent, dans ces exercices combinatoires, des expressions qui sont "presque" bonnes, à une application de List.map ou List.flatten près. C'est parce qu'on a tendance à "lifter" implicitement les expressions qui calculent une solution (combinaison, permutation, partition, etc) au cas où elles doivent en calculer plusieurs. Peut-on cacher toutes ces rustines combinatoires et obtenir du code plus lisible, plus concis et plus compréhensible ? Oui, en utilisant un concept construit exprès pour lifter des calculs en changeant (et en enrichissant) leur interprétation : les monades.

Plaçons-nous dans le cadre où on veut obtenir "gratuitement", ou de façon la plus transparente possible, l'ensemble des solutions d'un problème combinatoire donné. Il se trouve que calculer toutes les solutions ou bien en prendre une au hasard revient au même et la monade que nous allons définir s'appelle traditionnellement la monade non-déterministe NDET.

Pour construire une monade, il nous faudra d'abord un constructeur de types et une fonction map, c'est-à-dire un "foncteur" :

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

L'opération map respecte les identités suivantes, qui constituent sa spécification :

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

Puis on spécifie un certain nombre d'opérations destinées à cacher la tubulure (ici, l'obtention de toutes les solutions), ce qui constitue une "monade".

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

Ces opérations elles-aussi respectent certaines identités, qui constituent leur spécification :

```
map f = bind (return ∘ f)
bind return = id
(bind f) ∘ return = f
(bind f) ∘ (bind g) = bind ((bind f) ∘ g)
```

On peut remarquer que l'utilisation d'un type spécifique 'a t évite les confusions entre les listes, les listes de listes, etc. L'opération return permet de transformer un calcul ordinaire dont le résultat est unique en un ensemble de solutions (à un élément bien sûr). L'opération bind permet d'enchaîner les calculs en combinant (ici en applatissant) les ensembles de solutions à chaque étape correspondant à une application de f. Les équations (2) et (3) prouvent que return est un élément neutre pour bind, en ses deux arguments. La (4) prouve qu'enchaîner les calculs de solutions de f et ceux de g revient "presque" à enchaîner les calculs de f ∘ g, à un bind près.

Remarque : C'est ce même "presque" qu'on oublie souvent quand on combine les solutions partielles manuellement, sans monade.

▷ Support étudiant

Cette spécification équationnelle peut servir à transformer/optimiser des programmes, faire des preuves de programmes, ou encore à proposer des jeux de tests pour vérifier que l'implantation choisie forme bien une monade.

On pourrait définir notre monade en termes ensemblistes, puisque l'on calcule un ensemble de solutions :

$$return\ a = \{a\}$$
$$bind\ f\ ens = \bigcup_{a \in ens} f(a)$$

En OCaml, cela donne, en supposant que la fonction f est injective, ce qui sera le cas pour le calcul des permutations (sinon il faut éliminer les doublons) :

```
module NDET : MONADE =
struct
type 'a t = 'a list

let map = List.map

let return a = [a]

let bind f ens = List.flatten (List.map f ens)
end
```

La fonction permutations : 'a list -> 'a list NDET.t, en supposant insertions : 'a -> 'a list -> 'a list NDET.t s'écrirait maintenant :

```
open NDET
let rec permutations l =
```

```
 match l with
 | []     -> return []
 | e::q -> bind (insertions e)  (permutations q)
```

Remarque : L'usage du bind, surtout imbriqué, ne le rend pas très lisible. On a tendance en pratique à le remplacer par un opérateur infixe >>= : 'a t -> ('a -> 'b t) -> 'b t. On aurait alors :

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

ou encore, plus explicitement :

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

Cette implantation respecte bien la spécification proposée mais ceci n'est pas suffisant, puisqu'on ne peut engendrer que des singletons (avec return) ! En les combinant entre eux, on n'obtiendra jamais que d'autres singletons... On doit donc ajouter deux opérations à cette monade pour pouvoir faire quelque chose, on la transforme alors en monade dite "additive".

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

Ces deux opérations forment un monoïde et bind est linéaire en ses deux arguments :

```
zero ++ a = a ++ zero= a
(a ++ b) ++ c = a ++ (b ++ c)

(bind f) zero = zero
(bind f) (a ++ b) = (bind f a) ++ (bind f b)
bind (fun _ -> zero) = (fun _ -> zero)
bind (fun x -> f x ++ g x) = fun x -> (bind f) x ++ (bind g) x
```

L'opération zero, donne l'ensemble vide de solutions (quelquefois appelé fail). L'opération ++ est l'union des ensembles de solutions. En OCaml, cela donne pour notre monade, en supposant toujours qu'il n'y a pas de doublons :

```
module NDET : MONADE_PLUS =
struct
⋮
let zero = []

let (++) = (@)
end
```

On peut maintenant écrire insertions : 'a -> 'a list -> 'a list NDET.t :

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

1.1 Autres monades

On va voir que le concept de monade permet de représenter des phénomènes apparemment très différents.

▷ **Support étudiant**

Pour montrer toute la puissance du concept de monade, on propose une autre implantation de notre monade additive qui permet de transformer la génération exhaustive de toutes les solutions, très coûteuse en mémoire pour des problèmes combinatoires, en génération itérative, à l'aide d'un itérateur qui calcule les solutions les unes après les autres, jusqu'à la dernière.

▷ **Exercice 1** *On souhaite une autre implantation de la monade* NDET*, qui réalise un accès à la demande des solutions, en utilisant les constructions paresseuses du module* Lazy*. On utilisera le type "liste paresseuse" suivant :*

```
type 'a t    = Iter of 'a node Lazy.t
and 'a node = ('a * 'a t) option
```

▷ ***Solution***

La définition est celle des flux, sauf que dans notre cas, ces flux sont nécessairement finis, d'où le nom "liste paresseuse". La fonction all_solutions *n'aurait pas de sens pour des flux. On se contente de retarder l'évaluation à l'aide des constructions paresseuses. Il est utile de faire écrire* uncons *pour simplifier les écritures.*

```
(* fonction auxiliaire uncons *)
let uncons (Iter iter) = Lazy.force iter
let rec map f iter =
  Iter (lazy (match uncons iter with
        | None           -> None
        | Some (e, iter') -> Some (f e, map f iter')))
let return a =
  Iter (lazy (Some (a, Iter (lazy None))))
let zero =
  Iter (lazy None)
let rec (++) iter1 iter2 =
  Iter (lazy (match uncons iter1 with
        | None           -> uncons iter2
        | Some (e, iter1') -> Some (e, iter1' ++ iter2)))
let rec bind f iter =
  Iter (lazy (match uncons iter with
        | None           -> None
        | Some (e, iter') -> uncons (f e ++ bind f iter')))
let rec all_solutions (Iter iter) =
  match Lazy.force iter with
  | None           -> []
  | Some (e, iter') -> e::(all_solutions iter')
```

Toujours à l'aide des flux/listes paresseuses, on souhaire maintenant définir la monade WRITER. L'effet de cette monade, modélisé par le type 'a WRITER.t est de produire un flux de sortie en même temps qu'une valeur de type 'a. Cette monade est en fait paramétrée par le type (constant) des éléments des flux produits. Ce flux de sortie permet de loguer les événements importants du calcul, à des fins de *monitoring/debugging* par exemple. Ce mécanisme est représenté par la présence d'une fonction tell : W.t -> unit WRITER.t, où W.t est le type des éléments des flux.

▷ **Exercice 2** *Définir la monade* WRITER*, paramétrée par un module* (W : **sig type** t **end**).

▷ ***Solution***

Le type est fonctoriel en 'a*, le type des éléments du flux de sortie étant constant. On définit en plus la fonction* run : 'a WRITER.t -> 'a * W.tFlux.t *qui permet de récupérer la valeur résultat et le flux de sortie produit.*

```
module WRITER (W : sig type t end) (*: MONADE*) =
struct
  type 'a t = 'a * W.t Flux.t

  let return a = (a, Flux.vide)

  let map f (a, w) = (f a, w)

  let ( >>= ) (a, w) f = let (b, w') = f a in (b, Flux.append w w')

  let tell s = ((), Flux.(cons s vide))

  let run (v, w) = (v, w)
end
```

▷ **Exercice 3** *Instrumenter en style monadique le programme suivant, à l'aide de la monade* WRITER*, afin de loguer les appels récursifs effectués (valeur du paramètre).*

```
let rec collatz n =
  if n = 1 then 1 else
  if n mod 2 = 0 then collatz (n / 2) else collatz (3 * n + 1)
```

▷ ***Solution***

```
module Int =
struct
  type t = int
end
module Wri = WRITER (Int)
let collatz n =
  let rec collatz_wri n =
    Wri.( tell n >>= fun () ->
    if n = 1 then return 1 else
    if n mod 2 = 0 then collatz_wri (n / 2) else collatz_wri (3 * n + 1))
  in collatz_wri n
```

On peut écrire un programme principal qui affiche les valeurs successives du paramètre (on devrait plutôt faire un joli graphique, mais bon). On suppose avoir défini Flux.iter .

```
let main n =
```

```
let (v, w) = Wri.(run (collatz n)) in
Flux.iter (fun i -> Format.printf "%d" i) w
```

▷ **Support étudiant**

En conclusion, l'articulation du code autour de la notion de monade permet ici très simplement et modulairement : de construire toutes les permutations, de les obtenir à la demande, de les tirer au sort, de les évaluer toutes dans des mondes différents, etc. Le concept de monade se prête à de nombreux autres usages et permet de représenter proprement, au niveau des types, divers mécanismes et effets de bord, comme le non-déterminisme, les exceptions, les entrées-sorties, les variables modifiables, etc.

Une autre question est la suivante : peut-on trouver des monades dans les langages de programmation "mainstream" ? Oui, en partie, car de nombreux langages permettent d'annoter et de vérifier l'usage des termes, fonctions, procédures, etc, en fonction des effets auxquels correspond leur évaluation et ainsi de représenter d'une certaine façon ces effets dans le système de types. On peut penser au modificateur de type `const` ou à la déclaration obligatoire des exceptions dans l'interface des méthodes en JAVA.

▷ **Support étudiant**

1.2 Encore une autre monade : la monade d'état

La monade d'état permet de représenter une valeur modifiable, un "état global", dans un langage fonctionnel. Ce mécanisme est très utilisé dans les langages purs, i.e. sans effets de bords tels que l'affectation, par exemple en Haskell. Le langage OCaml dispose déjà d'effets de bords comme les variable mutables et les affectations, ce qui diminue l'intérêt de cette monade.

Bien sûr, comme pour la monade NDET, de nombreuses implantations sont possibles de façon très modulaire, de la variable mutable ordinaire à la donnée distante accessible par messages, en passant par le flux de toutes les affectations successives pour pouvoir les annuler ou les loguer, etc. La monade STATE est une monade simple, non additive. Le type `'a t` représente les valeurs de type `'a` obtenue en réalisant un calcul sur un état de type `S.t` fixé. La monade STATE est elle aussi paramétrée par le type `S.t`. En tant qu'état, STATE possède également deux opérations supplémentaires classiques : `get : S.t t` et `put : S.t -> unit t`.

Enfin, comme dans d'autres monades, on a besoin d'encapsuler les calculs monadiques effectués, afin de récupérer une valeur "normale" en fin de calcul, ce qui permet de communiquer avec le reste d'une application qui n'aurait pas besoin de monade de façon permanente. Ceci est réalisé dans la monade STATE par `run : 'a t -> S.t -> 'a`, avec comme arguments le programme avec effets monadiques et (une valeur de) l'état initial de type `S.t` et un résultat final de type `'a`.

On donne la sémantique de la monade au moyen des triplets de Hoare suivants, où "ret" est la variable représentant le résultat du calcul, tandis que "st" représente l'état :

$$\frac{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathit{calcul}\ \{\mathrm{ret} = r,\ \mathrm{st} = \mathit{spost}\,\}}{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathrm{map}\ f\ \mathit{calcul}\ \{\,\mathrm{ret} = (f\ r),\ \mathrm{st} = \mathit{spost}\,\}} \qquad \frac{}{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathrm{return}\ a\ \{\,\mathrm{ret} = a,\ \mathrm{st} = \mathit{spre}\,\}}$$

$$\frac{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathit{calcul}_1\ \{\mathrm{ret} = r_1,\ \mathrm{st} = \mathit{smid}\,\} \qquad \{\,\mathrm{r} = r_1,\ \mathrm{st} = \mathit{smid}\,\}\ \mathit{calcul}_2\ \{\mathrm{ret} = r_2,\ \mathrm{st} = \mathit{spost}\,\}}{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathit{calcul}_1\ >\!\!>\!\!=\ (\mathrm{fun\ r} \rightarrow \mathit{calcul}_2)\ \{\,\mathrm{ret} = r_2,\ \mathrm{st} = \mathit{spost}\,\}}$$

$$\frac{}{\{\,\mathrm{st} = \mathit{spre}\,\}\ \mathrm{get}\ \{\,\mathrm{ret} = \mathit{spre},\ \mathrm{st} = \mathit{spre}\,\}} \qquad \frac{}{\{\,\mathit{true}\,\}\ \mathrm{put}\ \mathit{spost}\ \{\,\mathrm{ret} = (),\ \mathrm{st} = \mathit{spost}\,\}}$$

▷ **Exercice 4** *Définir la monade* STATE*, paramétrée par un module* (S : **sig type** t **end**).

▷ ***Solution***

Les préconditions sont des paramètres, les postconditions des résultats.

```
module STATE (S : sig type t end) (* : MONADE *) =
struct
type 'a t = S.t -> 'a * S.t
let map f calcul = fun spre -> let (retour, spost) = calcul spre in (f retour, spost)
let return a = fun spre -> (a, spre)
let (>>=) calcul f = fun spre -> let (retour, smid) = calcul spre in f retour smid
let get = fun spre -> (spre, spre)
let put spost = fun _ -> ((), spost)
let run calcul init = fst (calcul init)
end
```

On peut également utiliser de "vrais" effets de bords, en paramétrant par une référence de type S.t ref*. L'implantation est plus légère, essentiellement une monade* READER *en fait où la référence est l'*input *implicite.*

```
module STATE (S : sig type t end) (* : MONADE *) =
struct
type 'a t = S.t ref -> 'a
let map f calcul = fun sref -> f (calcul sref)
let return a = fun sref -> a
let (>>=) calcul f = fun sref -> f (calcul sref) sref
let get = fun sref -> !sref
let put spost = fun sref -> (sref := spost; ())
let run calcul init = calcul (ref init)
end
```

▷ **Exercice 5** *Utiliser la monade* STATE *pour lire une expression au clavier et vérifier uniquement qu'elle est bien parenthésée, à l'aide d'une variable d'état entière. On utilisera le canevas suivant :*

```
module Int =
struct
  type t = int
end
module St = STATE (Int)
(* well_paren : in_channel -> bool St.t *)
let rec well_paren f =
  St.(
  ...
  try
    (match input_char f with
     ...
    )
  with End_of_file -> ...)
```

▷ ***Solution***

Il faut passer outre l'aspect peu élégant du problème, où l'on manipule en même temps des vrais effets de bords (lecture dans un fichier) et des effets de bords monadiques.

Ici, au-delà de l'aspect syntaxique un peu lourd de la monade d'état, qu'on pourrait soulager avec des notations appropriées rappelant les langages impératifs (par des extensions de OCaml*), il faut souligner qu'on a une solution où l'on a fait disparaître syntaxiquement l'état (le compteur de parenthèses), réduisant ainsi le nombre*

de paramètres explicites de la fonction well_paren*. L'avantage serait d'autant plus important si notre code devait passer ce compteur de fonctions en fonctions, ce qui est souvent le cas en programmation fonctionnelle et diminue la lisibilité du code.*

```
let rec well_paren f =
  St.(
  get >>= fun s -> if s < 0 then return false else
  try
    (match input_char f with
    | '(' -> (map succ get) >>= put
    | ')' -> (map pred get) >>= put
    | _   -> return ()
    ) >>= (fun () -> well_paren f)
  with End_of_file -> map ((=) 0) get);;

let main () =
  St.run (well_paren stdin) 0;;
```

L'usage de map *n'est pas recommandé car l'ordre des arguments n'est pas cohérent avec celui de* >>=*, ce qui peut être perturbant. On peut préférer écrire une solution avec plus d'arguments explicites :*

```
let rec well_paren f =
  St.(
  get >>= fun s -> if s < 0 then return false else
  try
    (match input_char f with
    | '(' -> get >>= (fun s -> put (succ s))
    | ')' -> get >>= (fun s -> put (pred s))
    | _   -> return ()
    ) >>= (fun () -> well_paren f)
  with End_of_file -> get >>= (fun s -> return (s=0)));;
```

Enfin, si l'on voulait à la fois vérifier les parenthèses et compter les retours à la ligne par exemple, le plus simple serait de manipuler une variable d'état global de type int * int*. L'autre solution est de définir la bonne composition de monades.*

1.3 Monades généralisées

Une généralisation des monades consiste à mélanger *type-state* et monade, ce qui permet de dévoiler une partie choisie de l'état interne d'un effet monadique et ainsi de décrire des machines à états. Une monade (additive) généralisée se présente ainsi :

Ici, ('i, 'o, 'a) t représente une valeur de type 'a dont l'effet passe de l'état 'i à 'o lorsqu'on l'évalue.

On peut par exemple utiliser cette spécification pour imposer un usage linéaire de ressources, dans la mesure où on n'utilise que la monade généralisé pour coder et notamment le bind, à l'exclusion de tout "**;**". En effet, OCaml comportant des effets de bord natifs non contrôlés, le mélange de ceux-ci avec ceux introduits par l'utilisateur sous forme de monades généralisées produit du code qui risque de ne pas respecter la sémantique attendue des monades, et donc, dans notre cas de figure, de ne pas garantir un usage linéaire des ressources.

Pour continuer l'exemple de la monade d'"état, on souhaite par exemple s'assurer statiquement qu'on ne pourra "écrire" dans la variable d'état qu'une fois et une seule (sans compter la valeur initiale). C'est une spécification qu'on va retrouver fréquemment dans des codes impératifs et qui est difficile à vérifier.

▷ **Exercice 6** *Définir la signature* LIN_ST_MONADE *d'une monade d'état généralisée, qui garantit une écri-*

ture unique, puis implanter cette signature, en utilisant la monade STATE.

▷ ***Solution***

Toute la sémantique de la linéarité est contenue dans la signature LIN_ST_MONADE. *Tout d'abord, le type* ('i, 'o, 's, 'a) t *est la juxtapostion de la monade d'état et d'une monade généralisée. Puis on définit les valeurs pertinentes pour les types* 'i *et* 'o, *qui seront contrôlées lorsqu'on voudra imposer l'écriture unique. Deux types (fantômes) particuliers suffisent :* init *pour l'état initial et* written *pour l'état après la première (et unique) écriture. Ces deux types sont définis en tant que types abstraits, i.e. sans équation.*

L'implantation est identique à STATE, *seuls les types changent pour les opérations* get, put *et* run, *spécifiques à la monade d'état.* get *n'a aucune contrainte sur l'état de la monade et peut être appelé n'importe quand.* put *fait passer l'état de* init *à* written, *ce qui ne peut avoir lieu au plus qu'une seule fois. Enfin,* run *impose qu'à travers le code avec effets monadiques, l'état passe effectivement de* init *à* written.

```
module type LIN_ST_MONADE =
sig
  type init
  type written
  type ('i, 'o, 's, 'a) t

  val map : ('a -> 'b) -> ('i, 'o, 's, 'a) t -> ('i, 'o, 's, 'b) t
  val return : 'a -> ('i, 'i, 's, 'a) t
  val (>>=) : ('i, 'm, 's, 'a) t -> ('a -> ('m, 'o, 's, 'b) t) -> ('i, 'o, 's, 'b) t
  val get : ('i, 'i, 's, 's) t
  val put : 's -> (init, written, 's, unit) t
  val run : (init, written, 's, 'a) t -> 's -> 'a
end

module LINEAR_STATE : LIN_ST_MONADE =
struct
  type init
  type written
  type ('i, 'o, 's, 'a) t = ('s, 'a) STATE.t

  include (STATE : ST_MONADE with type ('s, 'a) t := ('s, 'a) STATE.t)
end
```

Ci-dessous quelques exemples de code non-linéaire rejetés par le système de types.

```
LINEAR_STATE.(
    run (put 1 >>= (fun () -> get >>= (fun s -> put (succ s)))) 0
);;
LINEAR_STATE.(
    run (get >>= (fun s -> return (s=0))) 0
);;
```

Remarque : On peut également représenter les continuations délimitées, abordées dans un prochain TD, dans le cadre des monades généralisées.

1.4 Composition de monades

On voudrait naturellement pouvoir composer des monades pour en combiner les effets, *i.e.* pouvoir modéliser à la fois des exceptions et du non-déterminisme ou autre. Malheureusement, les monades se combinent assez mal. Sur un plan théorique, c'est parce qu'elles représentent chacune le "repliage" d'une adjonction, *i.e.*

la composition des deux opérateurs de cette adjonction (qui ne forme pas nécessairement l'identité). Pour composer deux monades de façon générale, il faudrait disposer des adjonctions dont elles sont respectivement issues, ce qui n'est pas le cas en partique, pour pouvoir recomposer ces adjonctions. Néanmoins, certains mécanismes particuliers existent comme la pile de monades, la promotion (lifting) de monades, etc. Rien de très général toutefois. C'est la grosse limitation de cette approche monadique. D'autres constructions plus récentes comme les monades "libres" ou les effets algébriques pallient ces problèmes.