TD7 : Structures monadiques

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 $f \circ g$, à un bind près.

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 :

$$\begin{aligned} &return\ a = \{a\} \\ &bind\ f\ ens = \bigcup_{a \in ens} f(a) \end{aligned}$$

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.

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
```

Toujours à l'aide des flux/listes paresseuses, on souhaite 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**).

▷ **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)
```

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.

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 :

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

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

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

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

▷ **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 -> ...)
```

1.3 Monades généralisées

▷ **Exercice 6** *Définir la signature* LIN_ST_MONADE *d'une monade d'état généralisée, qui garantit une écriture unique, puis implanter cette signature, en utilisant la monade* STATE*.*