Programmation Fonctionnelle Avancée

Séance 5 : Monads, Maybe, Writer

Alexandros Singh

Université Paris 8

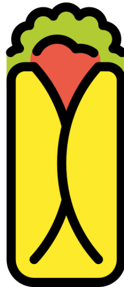
9 novembre 2023

Une monade est...

Une monade est... *juste* un monoïde dans la catégorie des endofunctors !

C'est quoi le problème ?

Une monade est... comme un burrito !



Une monade est... une structure qui fournit :

- un type paramétré `'a t`, dont les valeurs sont appelées *valeurs monadiques*,
- une fonction `return` pour de créer des valeurs de type `'a t` à partir de valeurs de type `'a`,
- et une fonction `bind` pour appliquer des fonctions de type `'a -> 'b t` aux valeurs de type `'a t`, produisant une valeur de type `'b t`.

En OCaml, une monade est une structure conforme à la signature suivante :

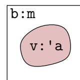
```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

et qui respecte certaines lois que nous préciserons plus tard.

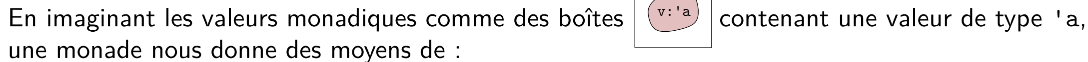
La fonction `bind` étant souvent utilisée en infixe, nous allons la remplacer par un opérateur »= (toujours prononcé comme “bind”).

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

L’allégorie des boîtes

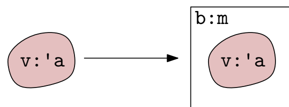
En imaginant les valeurs monadiques comme des boîtes 

contenant une valeur de type `'a`, une monade nous donne des moyens de :

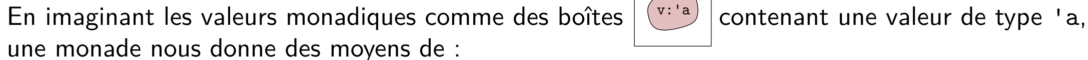


En imaginant les valeurs monadiques comme des boîtes contenant une valeur de type `'a`, une monade nous donne des moyens de :

- `return` : mettre une valeur de type `'a` dans une boîte `'a t`,

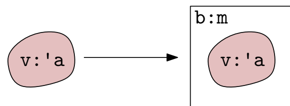


L’allégorie des boîtes

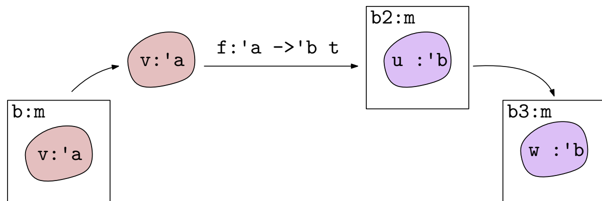
En imaginant les valeurs monadiques comme des boîtes 

contenant une valeur de type `'a`, une monade nous donne des moyens de :

- `return` : mettre une valeur de type `'a` dans une boîte `'a t`,



- `bind` : extraire une valeur d’une boîte `'a t`, lui appliquer une fonction produisant une nouvelle valeur encapsulée `’ b t`, et enfin retourner une valeur encapsulée `'b t`.



Notez que la valeur monadique `b3:'b m` peut “dépendre” à la fois de `b2:'b m` mais aussi de `b:textquotesingle a m`.

Nous avons déjà utilisé l'allégorie de la boîte en parlant du module `Option`. Ce n'est pas une coïncidence !

- Le module Option possède tous les ingrédients pour former un type de monade appelé la monade Maybe, qui est utilisé pour représenter les calculs qui peuvent résulter en "rien".
- Plus formellement, cela nous permet de travailler avec des *fonctions partielles* qui ne sont pas définies pour toutes les valeurs d'entrée.

Nous allons maintenant examiner un exemple d'utilisation d'une telle monade.

Supposons que nous voulions implémenter une fonction `nth_to_str` qui convertit le n-ième élément d'une liste donnée d'entiers en une chaîne de caractères. Le comportement attendu est le suivant :

```
# nth_to_str [1;2;3;4] 0;;
- : string = "1"
# nth_to_str [1;2;3;4] 1;;
- : string = "2"
# nth_to_str [1;2;3;4] 2;;
- : string = "3"
# nth_to_str [1;2;3;4] 3;;
- : string = "4"
```

Une telle fonction peut être réalisée en composant une fonction qui obtient le n-ième élément d'une liste avec la fonction `string_of_int` qui convertit les entiers en chaînes de caractères.

```
# let nth_to_str l n = (List.nth l n) |> string_of_int;;
val nth_to_str : int list -> int -> string = <fun>
# nth_to_str [1;2;3] 0;;
- : string = "1"
# nth_to_str [1;2;3] 1;;
- : string = "2"
# nth_to_str [1;2;3] 2;;
- : string = "3"
```

Mais que se passe-t-il si nous appelons `nth_to_str [1;2;3] 4` ? Ou `nth_to_str [1;2;3] -10` ?

```
# nth_to_str [1;2;3] 4;;
Exception: Failure "nth".
# nth_to_str [1;2;3] (-10);;
Exception: Invalid_argument "List.nth".
```

La monade Maybe - exemple

Mais que se passe-t-il si nous appelons `nth_to_str [1;2;3] 4` ? Ou `nth_to_str [1;2;3] -10` ?

```
# nth_to_str [1;2;3] 4;;
Exception: Failure "nth".
# nth_to_str [1;2;3] (-10);;
Exception: Invalid_argument "List.nth".
```

La fonction lève une erreur, car `Liste.nth` est partiellement défini : le n-ième élément d'une liste de longueur inférieure à n n'est pas défini ! Idem pour n négatif !

La monade Maybe - exemple

Mais que se passe-t-il si nous appelons `nth_to_str [1;2;3] 4` ? Ou `nth_to_str [1;2;3] -10` ?

```
# nth_to_str [1;2;3] 4;;
Exception: Failure "nth".
# nth_to_str [1;2;3] (-10);;
Exception: Invalid_argument "List.nth".
```

La fonction lève une erreur, car `Liste.nth` est partiellement défini : le n-ième élément d'une liste de longueur inférieure à n n'est pas défini ! Idem pour n négatif !
Cette façon de gérer les erreurs pose quelques difficultés :

- La fonction `nth_to_str` est de type `int list -> int -> string`. Remarquez comment le type ne nous dit pas que cette fonction pourrait "échouer" !

La monade Maybe - exemple

Mais que se passe-t-il si nous appelons `nth_to_str [1;2;3] 4` ? Ou `nth_to_str [1;2;3] -10` ?

```
# nth_to_str [1;2;3] 4;;
Exception: Failure "nth".
# nth_to_str [1;2;3] (-10);;
Exception: Invalid_argument "List.nth".
```

La fonction lève une erreur, car `Liste.nth` est partiellement défini : le n-ième élément d'une liste de longueur inférieure à n n'est pas défini ! Idem pour n négatif !

Cette façon de gérer les erreurs pose quelques difficultés :

- La fonction `nth_to_str` est de type `int list -> int -> string`. Remarquez comment le type ne nous dit pas que cette fonction pourrait "échouer" !
- Si elle est utilisée dans un programme sans gestion appropriée des erreurs (c'est-à-dire en dehors d'un "`try-with`"), l'appel à cette fonction peut entraîner un crash !

Une autre façon de gérer cette situation problématique est d'utiliser l'alternative suivante à `Liste.nth` (voir aussi `List.nth_opt`) :

```
let nth_t l n =
  if n < 0 then None else
  let rec nth_aux l n =
    match l with
    | [] -> None
    | a::l -> if n = 0 then Some a else nth_aux l (n-1)
  in nth_aux l n
```

Cette fonction est totale : la possibilité d'échouer est explicitement représentée par le fait qu'elle peut renvoyer `None` !

Mais cela a créé un nouveau problème, nous ne pouvons plus composer nos deux fonctions :

```
# let nth_to_str_attempt l n = (nth_t l n) |> string_of_int;;
Error: This expression has type 'a option
       but an expression was expected of type int
```

Mais cela a créé un nouveau problème, nous ne pouvons plus composer nos deux fonctions :

```
# let nth_to_str_attempt l n = (nth_t l n) |> string_of_int;;
Error: This expression has type 'a option
       but an expression was expected of type int
```

- `List.nth_opt` renvoie une valeure de type `int` enveloppée (comme dans une boîte !) par `option`.

Mais cela a créé un nouveau problème, nous ne pouvons plus composer nos deux fonctions :

```
# let nth_to_str_attempt l n = (nth_t l n) |> string_of_int;;
Error: This expression has type 'a option
       but an expression was expected of type int
```

- `List.nth_opt` renvoie une valeure de type `int` enveloppée (comme dans une boîte !) par `option`.
- `string_of_int` attend une valeure de type `int` et produit une chaîne de caractères.

Mais cela a créé un nouveau problème, nous ne pouvons plus composer nos deux fonctions :

```
# let nth_to_str_attempt l n = (nth_t l n) |> string_of_int;;
Error: This expression has type 'a option
       but an expression was expected of type int
```

- `List.nth_opt` renvoie une valeure de type `int` enveloppée (comme dans une boîte !) par `option`.
- `string_of_int` attend une valeure de type `int` et produit une chaîne de caractères.

Les deux ne sont plus compatibles !

Corriger ce problème est fastidieux mais assez facile :

```
let nth_to_str l n = match (nth_t l n) with
    | None -> None
    | Some x -> Some (string_of_int x)
```

Corriger ce problème est fastidieux mais assez facile :

```
let nth_to_str l n = match (nth_t l n) with
    | None -> None
    | Some x -> Some (string_of_int x)
```

- Notez que nous renvoyons une valeur de type option, afin de rester explicite sur la possibilité d'un échec.

La monade Maybe - exemple

Corriger ce problème est fastidieux mais assez facile :

```
let nth_to_str l n = match (nth_t l n) with
    | None -> None
    | Some x -> Some (string_of_int x)
```

- Notez que nous renvoyons une valeur de type option, afin de rester explicite sur la possibilité d'un échec.
- Et si nous voulions composer ce qui précède avec `print_endline` afin d'imprimer le résultat final (s'il existe) ?

La monade Maybe - exemple

Corriger ce problème est fastidieux mais assez facile :

```
let nth_to_str l n = match (nth_t l n) with
    | None -> None
    | Some x -> Some (string_of_int x)
```

- Notez que nous renvoyons une valeur de type option, afin de rester explicite sur la possibilité d'un échec.
- Et si nous voulions composer ce qui précède avec `print_endline` afin d'imprimer le résultat final (s'il existe) ?

```
let print_nth l n = match (nth_to_str l n) with
    | None -> None
    | Some x -> Some (print_endline x)
```

Un motif émerge ! Pour “composer” une fonction `f` qui renvoie une valeur de type `'a option` avec une fonction `g : 'a -> 'b`, afin d'obtenir `h : 'a option -> 'b option` :

Un motif émerge ! Pour “composer” une fonction `f` qui renvoie une valeur de type `'a option` avec une fonction `g : 'a -> 'b`, afin d'obtenir `h : 'a option -> 'b option` :

- “déballer” le résultat de `f` :

La monad Maybe - exemple

Un motif émerge ! Pour “composer” une fonction `f` qui renvoie une valeur de type `'a option` avec une fonction `g : 'a -> 'b`, afin d'obtenir `h : 'a option -> 'b option` :

- “déballer” le résultat de `f` :

```
let h x = match (f x) with
    | None -> None
    | Some -> (* quelque chose qui implique g *)
```

La monad Maybe - exemple

Un motif émerge ! Pour "composer" une fonction `f` qui renvoie une valeur de type `'a option` avec une fonction `g : 'a -> 'b`, afin d'obtenir `h : 'a option -> 'b option` :

- "déballer" le résultat de `f` :

```
let h x = match (f x) with
    | None -> None
    | Some -> (* quelque chose qui implique g *)
```

- le donner à `g` et "emballer" le résultat dans une "boîte standard" :

La monad Maybe - exemple

Un motif émerge ! Pour “composer” une fonction `f` qui renvoie une valeur de type `'a option` avec une fonction `g : 'a -> 'b`, afin d'obtenir `h : 'a option -> 'b option` :

- “déballer” le résultat de `f` :

```
let h x = match (f x) with
    | None -> None
    | Some r -> (* quelque chose qui implique g *)
```

- le donner à `g` et “emballer” le résultat dans une “boîte standard” :

```
let h x = match (f x) with
    | None -> None
    | Some r -> Some (g r)
```

Nous sommes maintenant prêts à définir notre implémentation de la monade Maybe :

Nous sommes maintenant prêts à définir notre implémentation de la monade Maybe :

- notre type monadique est `'a option` :

```
module Maybe : Monad  = struct
    type 'a t = 'a option
```

La monad Maybe - exemple

Nous sommes maintenant prêts à définir notre implémentation de la monade Maybe :

- notre type monadique est `'a option` :

```
module Maybe : Monad  = struct
    type 'a t = 'a option
```

- la fonction `return` (“emballage avec une boîte standard”) est :

```
    let return x = Some x
```

La monad Maybe - definition

Nous sommes maintenant prêts à définir notre implémentation de la monade Maybe :

- notre type monadique est `'a option` :

```
module Maybe : Monad  = struct
    type 'a t = 'a option
```

- la fonction `return` (“emballage avec une boîte standard”) est :

```
    let return x = Some x
```

- la fonction `bind` (“déballer” et donner le résultat à une fonction monadique) est :

```
    let ( >>= ) m f = match m with
        | None -> None
        | Some x -> f x
end
```

En utilisant cette monade, nous pouvons réécrire la fonction `nth_to_str` comme suit :

```
let str_of_int_m x = string_of_int x |> return

let nth_to_str_m l n =
    nth_t l n >>= fun a -> str_of_int_m a
```

En utilisant cette monade, nous pouvons réécrire la fonction `nth_to_str` comme suit :

```
let str_of_int_m x = string_of_int x |> return
let nth_to_str_m l n =
    nth_t l n >>= fun a -> str_of_int_m a
```

et la fonction `print_nth` comme :

```
let print_m x = print_endline x |> return
let nth_to_str_m_0 l n =
    nth_t l n >>= fun a ->
    str_of_int_m a >>= fun b ->
    print_m b
```

La monad Maybe - usage

Cette syntaxe peut rapidement devenir encombrante. Heureusement, OCaml (depuis la version 4.08) nous permet de définir des opérateurs `let` :

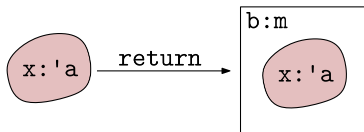
```
let (let*) x f = x >>= f

let nth_to_str_m_1 l n =
    let* a = nth_t l n in
    str_of_int_m a

let nth_to_str_m_1 l n =
    let* a = nth_t l n in
    let* b = str_of_int_m a in
    print_m b
```

Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`

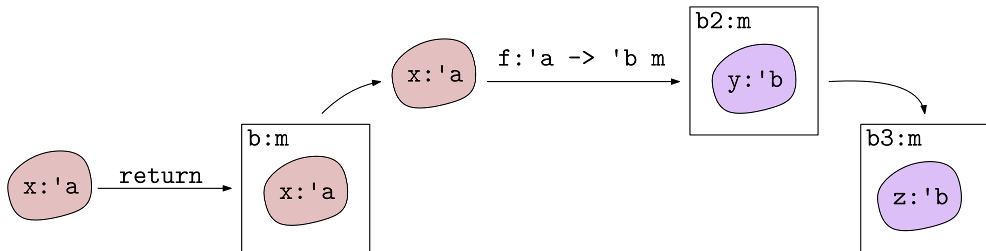
Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x` ≫= `f` se comporte comme `f x`



Toutes les monades doivent satisfaire aux trois conditions suivantes :

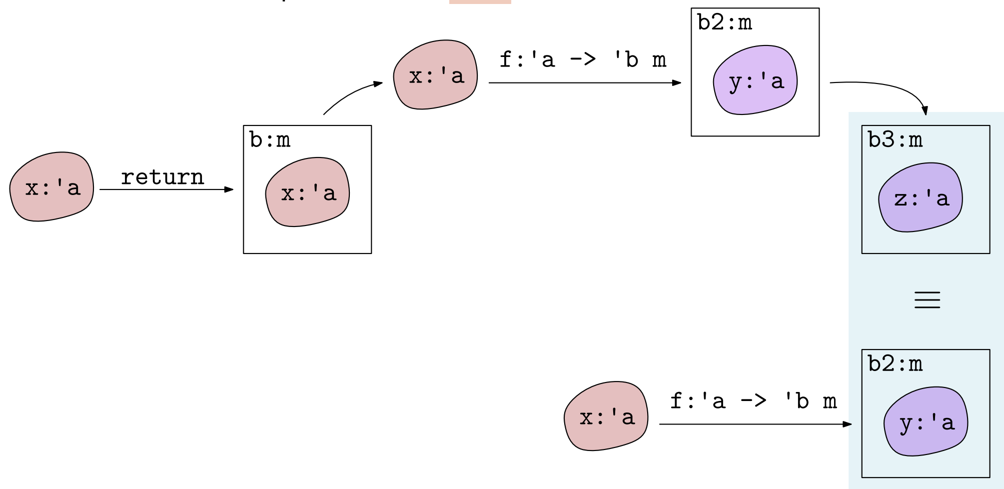
- `return x` `>>= f` se comporte comme `f x`



Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`



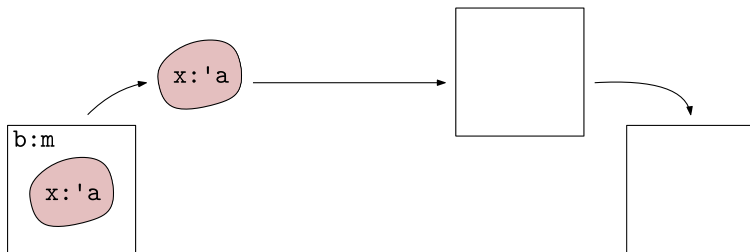
Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`

Toutes les monades doivent satisfaire aux trois conditions suivantes :

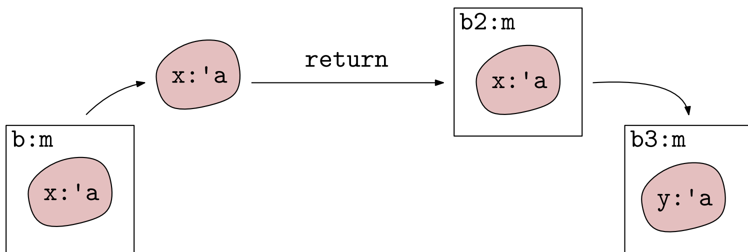
- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`



Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

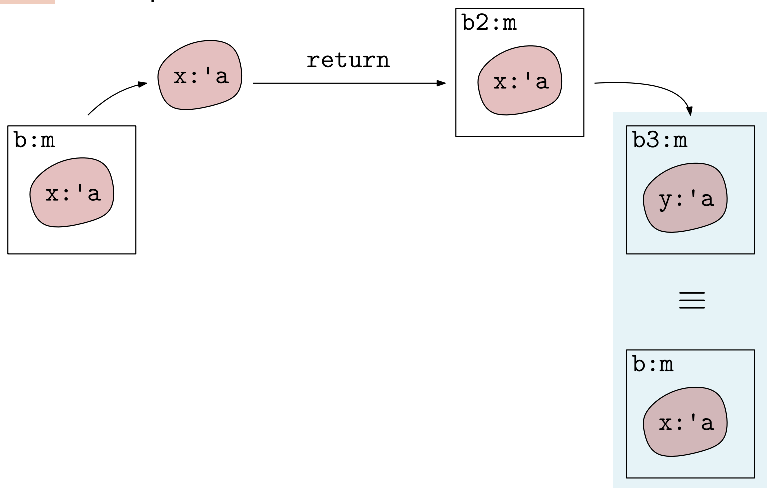
- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`



Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`



Les trois lois

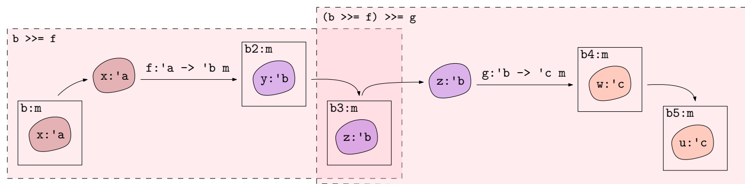
Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`,
- `(b >>= f) >>= g` se comporte comme `b >>= (fun x -> f x >>= g)`

Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

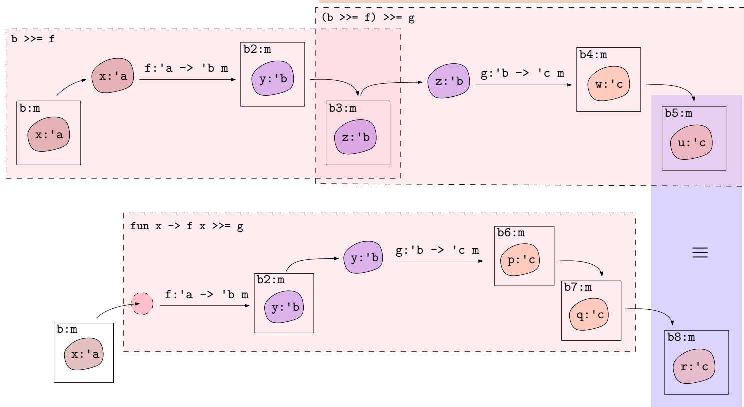
- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`,
- `(b >>= f) >>= g` se comporte comme `b >>= (fun x -> f x >>= g)`



Les trois lois

Toutes les monades doivent satisfaire aux trois conditions suivantes :

- `return x >>= f` se comporte comme `f x`,
- `b >>= return` se comporte comme `b`,
- `(b >>= f) >>= g` se comporte comme `b >>= (fun x -> f x >>= g)`



La troisième loi n'est pas facile à lire sous cette forme...
Définir l'opérateur >=> de composition de Kleisli comme suit :

```
let ( >=> ) f g x =
  f x >>= fun y ->
  g y
```

Kleisli

La troisième loi n'est pas facile à lire sous cette forme...
Définir l'opérateur >=> de composition de Kleisli comme suit :

```
let ( >=> ) f g x =
  f x >>= fun y ->
  g y
```

Cela permet de formuler autrement les trois lois :

- `return >=> f` se comporte comme `f`
- `f >=> return` se comporte comme `f`
- `(f >=> g) >=> h` se comporte comme `f >=> (g >=> h)`

Kleisli

La troisième loi n'est pas facile à lire sous cette forme...
Définir l'opérateur >=> de composition de Kleisli comme suit :

```
let ( >=> ) f g x =
  f x >>= fun y ->
  g y
```

Cela permet de formuler autrement les trois lois :

- `return >=> f` se comporte comme `f`
- `f >=> return` se comporte comme `f`
- `(f >=> g) >=> h` se comporte comme `f >=> (g >=> h)`

Une monade est un monoïde avec une opération binaire associative >=> et une identité `return` ! Et il enveloppe des choses - comme un burrito ?!