

# Programmation Fonctionnelle Avancée

## 10 : Monades

Pierre Letouzey

Université Paris Cité  
UFR Informatique  
Institut de Recherche en Informatique Fondamentale  
`letouzey@irif.fr`

13 avril 2022

## Qu'est-ce qu'une Monade ?

- ▶ À la base, une monade est un *type polymorphe*  $'a \rightarrow t$ , qu'on considère comme un *enrichissement* d'un type  $'a$ .
- ▶ Par exemple : passer d'un type  $'a$  à  $'a \rightarrow \text{option}$  afin de décrire de possibles absences de résultat (la monade d'erreur).
- ▶ Ou encore : passer de  $'a$  à  $'a \rightarrow \text{list}$  pour décrire un ensemble de résultats possibles (monade de listes).
- ▶ Ou encore : aller vers un type produit de  $'a$  avec quelque chose.
- ▶ Puis, il faut un moyen pour migrer les fonctions sur  $'a$  vers  $'a \rightarrow t$ .

## Interprètes extensibles

- ▶ La motivation originale pour les monades :
- ▶ Exploiter la programmation monadique...
- ▶ ... pour l'écriture d'interprètes extensibles

## Écrire un évaluateur

- ▶ L'exemple le plus ancien qui montre l'utilité des monades est celui d'un évaluateur pour un petit langage fonctionnel, auquel on ajoute petit à petit des traits supplémentaires : erreurs, exceptions, état, non déterminisme.
- ▶ Cet exemple est étroitement lié aux travaux originaux de Moggi sur la sémantique dénotationnelle.
- ▶ Petit langage d'expressions, avec des types `bool` et `int`, des expressions fonctionnelles et application de fonctions.
- ▶ Nous utilisons les variants polymorphes pour la syntaxe, pour pouvoir étendre plus facilement.

## Exemples (syntax.ml)

```
(* Syntaxe Abstraite *)
```

```
type id = string
```

```
and exp =
```

```
  [ `Int of int
  | `Bool of bool
  | `Var of id
  | `Op of op * exp * exp
  | `App of exp * exp
  | `Fun of id * exp
  | `If of exp * exp * exp ]
```

```
and op = Plus | Minus | Mult | Div | Eq | Lt | Gt
```

## Exemples (valeurs.ml)

```

(* The type of values.
   Values can be bound to identifiers. *)
type value =
  [ `VInt of int
  | `VBool of bool
  | `VFun of value -> result ]
(* The type of results. Results are obtained by *)
(* applying a function or operator to values. *)
and result = value

```

## Exemples (ops.ml)

*(\* Appliquer un opérateur ou une comparaison \*)*

```
let app_op o v1 v2 =
  match v1, v2 with
  | `VInt x, `VInt y ->
    (match o with
     | Plus -> `VInt (x + y)
     | Minus -> `VInt (x - y)
     | Mult -> `VInt (x * y)
     | Div -> `VInt (x / y)
     | Eq -> `VBool (x = y)
     | Lt -> `VBool (x < y)
     | Gt -> `VBool (x > y))
  | _ -> failwith "Operation on a non-integer"
```

# L'interpréteur I

```

let rec interp (exp:exp) env : result = match exp with
| `Var id      -> List.assoc id env
| `Int i       -> `VInt i
| `Bool b      -> `VBool b
| `Op(o,e1,e2) -> app_op o (interp e1 env) (interp e2 env)
| `If(b,e1,e2) ->
    (match interp b env with
    | `VBool true -> interp e1 env
    | `VBool false -> interp e2 env
    | _ -> failwith "Non_boolean_test")
| `App (e1,e2) ->
    (match interp e1 env with
    | `VFun f -> f (interp e2 env)
    | _ -> failwith "Application_of_non_function")
| `Fun (id,e) -> `VFun (fun a -> interp e ((id,a)::env))

```



## L'interpréteur II

```
(* Well typed program *)
let ok =
  interp (`App(`Fun("x", `If(`Var "x", `Int 1, `Int 2)),
    `Bool true)) []

(* Ill typed program *)
let ko = interp (`App(`Int 1, `Int 2)) [];;
```

## Interpréteur avec erreur

- ▶ On veut adapter l'interpréteur pour des programmes pouvant lever des erreurs.
- ▶ Quel impact sur le code ?
- ▶ En premier lieu, nous modifions le type du résultat, qui contient maintenant un constructeur d'erreur.
- ▶ Attention au cas de la fonction : nous écrivons un interpréteur en *appel par valeur*, donc le paramètre n'est jamais une erreur, parce que l'erreur aurait été capturée avant, lors de l'évaluation de l'argument.
- ▶ Chaque fois qu'on *utilise* le résultat d'un appel à l'interpréteur, on doit tester s'il s'agit d'une erreur, et le cas échéant la propager ☹.

## L'interpréteur avec gestion d'erreurs I

*(\* Interpreter with error handling \*)*

```

type value =
  [ `VInt of int | `VBool of bool
    | `VFun of value -> result_or_err ]
and result_or_err = Err | Val of value

let rec interperr exp env : result_or_err = match exp with
  | `Var(id)      -> Val (List.assoc id env)
  | `Int i        -> Val (`VInt i)
  | `Bool b       -> Val (`VBool b)
  | `Op(op,e1,e2) ->
    (match interperr e1 env with
     | Err -> Err
     | Val v1 ->
       (match interperr e2 env with
        | Err -> Err

```

## L'interpréteur avec gestion d'erreurs II

```

        | Val v2 -> Val (app_op op v1 v2)))
| `If(b,e1,e2) ->
    (match interperr b env with
    | Err -> Err
    | Val (`VBool true) -> interperr e1 env
    | Val (`VBool false) -> interperr e2 env
    | _ -> failwith "Non_boolean_test")
| `App (e1,e2) ->
    (match interperr e1 env with
    | Err -> Err
    | Val (`VFun f) ->
        (match interperr e2 env with
        | Err -> Err
        | Val v -> f v)
    | _ -> failwith "Application_of_non_function")
| `Fun (id,exp)->
    Val (`VFun (fun a -> interperr exp ((id,a)::env)))
    
```

## L'interpréteur avec gestion d'erreurs III

```
| `Fail -> Err
```

```
(* Program with no error *)
```

```
let _ = interperr
      (`App(`Fun("x",`If(`Var "x",`Int 1,`Int 2)),
            `Bool true)) []
```

```
(* interperr returns an error value *)
```

```
let _ = interperr
      (`App(`Fun("x",`If(`Var "x",`Fail,`Int 2)),
            `Bool true)) []
```

## Modifications nécessaire pour intégrer les erreurs

- ▶ On était obligé de distinguer le type des *valeurs* (qui peuvent par exemple être des arguments de fonctions) du type de *résultats* qui est plus riche.
- ▶ Attention, le type des résultats apparaît dans le type des valeurs (à cause des valeurs fonctionnelles).
- ▶ Ici, le type de résultat contient un “cas” supplémentaire.
- ▶ Il a fallu repenser l'utilisation d'un *résultat* en tant que *valeur* (application d'une fonction ou d'un opérateur).
- ▶ Parfois on veut considérer une valeur comme un résultat.

## Ajout d'un état

- ▶ Ajoutons maintenant une notion d'*état* à notre interpréteur : l'état sera ici une sorte de tableau infini indexé par des entiers et contenant des entiers.
- ▶ On représentera cela par une fonction `int -> int`.
- ▶ Le type de résultat change : c'est maintenant une paire d'une valeur et d'un état.
- ▶ L'interpréteur doit prendre un argument supplémentaire qui est un état.
- ▶ On pourrait admettre des valeurs générales dans les cases de notre tableau mais cela complexifie tout (cycles entre `state` et `value` et `result`).

```
(* The idealized imperative state we handle here :  
    an infinite array of integers , indexed by integers . *)
```

```
module type STATE = sig
```

```
  type t
```

```
  (* Initial state , containing 0 everywhere *)
```

```
  val init : t
```

```
  (* Reading the state at some position *)
```

```
  val get : t → int → int
```

```
  (* (set s i x) is s except that it contains x at i *)
```

```
  val set : t → int → int → t
```

```
end
```

```
(* A possible representation , via functions *)
```

```
module State : STATE = struct
```

```
  type t = int → int
```

```
  let init = fun _ → 0
```

```
  let get s i = s i
```

```
  let set s i x = fun j → if i=j then x else s j
```

```
end
```



## L'interpréteur avec état I

```
(* Imperative interpreter *)

(* Types of values and results *)
type state = State.t
type value_st =
  [ `VInt of int | `VBool of bool (* as before *)
  | `VFun of value_st -> state -> result_st
  | `VUnit (* result of imperative actions *) ]
and result_st = value_st * state

let rec interpst exp env (s:state) : result_st =
  match exp with
  | `Var id      -> (List.assoc id env), s
  | `Int i       -> `VInt i, s
  | `Bool b      -> `VBool b, s
  | `Op(o,e1,e2) ->
    let v1,s = interpst e1 env s in
```

## L'interpréteur avec état II

```

    let v2,s = interpst e2 env s in
    app_op o v1 v2, s
| `If(b,e1,e2) ->
    let v,s = interpst b env s in
    (match v with
     | `VBool true -> interpst e1 env s
     | `VBool false -> interpst e2 env s
     | _ -> failwith "Non_boolean_condition_in_conditional")
| `App (e1,e2) ->
    let v1,s = interpst e1 env s in
    (match v1 with
     | `VFun f -> let v2,s = interpst e2 env s in f v2 s
     | _ -> failwith "Non_functional_value_in_application")
| `Fun (id,e) ->
    `VFun (fun a -> interpst e ((id,a)::env)), s
(* Some new imperative expressions *)
| `Unit -> `VUnit, s
    
```

## L'interpréteur avec état III

```
| `Seq (e1,e2) ->
  let _,s = interpst e1 env s in interpst e2 env s
| `While (b,e) ->
  interpst (`If(b,`Seq(e,exp),`Unit)) env s
| `Get e ->
  let v,s = interpst e env s in
  (match v with
   | `VInt i -> `VInt (State.get s i), s
   | _ -> failwith "Not an integer in Get")
| `Set (e,e') ->
  let v,s = interpst e env s in
  (match v with
   | `VInt i ->
    let v',s = interpst e' env s in
    (match v' with
     | `VInt x -> `VUnit, State.set s i x
     | _ -> failwith "Not an integer in Set")
```

## L'interpréteur avec état IV

```
| _ -> failwith "Not an integer in Set")
```

*(\* Example of imperative program. Pseudo-code :  
while m[0]<10 do m[0]:=m[0]+1; m[1]:=m[1]+m[0] done; m[1] \*)*

```
let code_imp =
  let x = `Int 0 and y = `Int 1 in
  `Seq(`While(`Op(Lt,`Get x,`Int 10),
    `Seq(
      `Set(x,`Op(Plus,`Get x,`Int 1)),
      `Set(y,`Op(Plus,`Get y,`Get x)))),
    `Get y)
let _ = interpst code_imp [] State.init
```

# Tout change !

Nous avons dû *tout changer* dans le code de notre interpréteur.  
 Comparons le cas de l'application `App (e1,e2) :

```
(* langage simple *)
(match interp e1 env with
| `VFun f -> f (interp e2 env)
...

(* avec erreurs *)
(match interperr e1 env with
| Err -> Err
| Val (`VFun f) ->
  (match interperr e2 env with
  | Err -> Err
  | Val v -> f v)
...

(* avec etat *)
let v1,s = interpst e1 env s in
(match v1 with
| `VFun f -> let v2,s = interpst e2 env s in f v2 s
...

```

## Tout change !!

Même l'interprétation des valeurs (variables, fonctions) change !

```
(* langage simple *)
| `Fun (id,e) -> `VFun (fun a -> interp e ((id,a)::env))
| `Var id      -> List.assoc id env

(* avec erreurs *)
| `Fun (id,e) -> Val (`VFun (fun a ->
                             interperr e ((id,a)::env)))
| `Var id      -> Val (List.assoc id env)

(* avec etat *)
| `Fun (id,e) -> `VFun (fun a ->
                             interpst e ((id,a)::env)), s
| `Var id      -> (List.assoc id env), s
```

Comment faire mieux ?

## Les monades à la rescousse !

- ▶ A la base, l'idée est de distinguer un type de valeurs 'a d'un type représentant les *résultats d'un calcul* sur 'a.
- ▶ Une *monade* est constituée par :
  - ▶ un type polymorphe 'a t
  - ▶ une opération **bind** : 'a t  $\rightarrow$  ('a  $\rightarrow$  'b t)  $\rightarrow$  'b t
  - ▶ une opération **return** : 'a  $\rightarrow$  'a t
  - ▶ des équations que doivent satisfaire **bind** et **return**
- ▶ **bind** compose un *résultat de calcul* sur 'a, avec un consommateur d'une valeur de type 'a, lui-même produisant un nouveau résultat de calcul, cette fois-ci sur 'b.
- ▶ **bind** m f est fréquemment noté m  $\gg$  f comme en Haskell.
- ▶ **return** plonge une valeur dans un résultat de calcul.

## Propriétés des opérateurs de monade

Les trois équations suivantes doivent être satisfaites :

1.  $(m \gg= \text{return}) = m$
2.  $(\text{return } v \gg= f) = f \ v$
3.  $((m \gg= g) \gg= h) = m \gg= (\text{fun } x \rightarrow (g \ x \gg= h))$   
ceci étant une forme d'associativité du **bind**.



## Extensions

Une monade particulière peut fournir d'autres briques pour construire des calculs (à part de `bind` et `return`) :

- ▶ Il faut parfois une opération constante *zero* : *'a t* telle que *bind zero f = zero*. Cela sert par exemple avec les monades de listes.
- ▶ Pour les interpréteurs, nous allons ajouter une nouvelle opération *reveal* : *'a t -> 'a* qui permet d'extraire la valeur calculée, tout à la fin.

## L'évaluateur

Remarquons que nos interpréteurs sont des fonctions du type :

$$\text{interp} : \text{exp} \rightarrow (\text{id} * \text{value}) \text{ list} \rightarrow \text{result}$$

où :

- ▶ `exp` est le type des expressions à évaluer
- ▶ `id` est le type des identificateurs
- ▶ `value` est le type des valeurs qu'on peut associer à un identificateur
- ▶ `result` est le type des résultats retournés par l'interpréteur

## Généralisation

- Généralisons : `value` devient un paramètre `'a` et `result` une monade (`'a t`)
- Voici la table des types pour les trois cas :

langage	'a	'a t
simple	value	value
erreur	value	Err   Val of value
etat	value_st	state -> value_st * state

- Les monades utilisées, et leurs **return** :

monade T	'a t	return
identite	'a	fun v -> v
erreur	Err   Val of 'a	fun v -> Val v
etat	state -> 'a * state	fun v -> fun s -> v,s

## Utilisation de return

- Nous pouvons alors déjà unifier certains cas via **return** :

```
(* langage simple , erreurs , etat *)
| `Int i      -> return (`VInt i)
| `Bool b     -> return (`VBool b)
| `Var id     -> return (List.assoc id env)
| `Fun (id,e) -> return
                  (`VFun(fun a -> interp e ((id,a)::env)))
```

- Vérifiez qu'en utilisant la monade appropriée on ré-obtient bien le code correspondant aux interpréteurs précédents.

## Trouver le bon bind

Rappelons le code de l'application :

```
(* langage simple *)
(match interp e1 env with
| `VFun f -> f (interp e2 env)
...

(* avec erreurs *)
(match interperr e1 env with
| Err -> Err
| Val (`VFun f) ->
  (match interperr e2 env with
  | Err -> Err
  | Val v -> f v)
...

(* avec etat *)
let v1,s = interpst e1 env s in
(match v1 with
| `VFun f -> let v2,s = interpst e2 env s in f v2 s
...

```

Ici, reconnaître des **bind** est moins clair...

## Les bonnes définitions de bind

- Pour la monade identité, c'est juste une application :

```
type 'a t = a
let bind (m : 'a t) (f : 'a -> 'b t) : 'b t = m |> f
```

- Pour la monade d'erreur, il faut propager l'erreur :

```
type 'a t = Err | Val of 'a
let bind (m : 'a t) (f : 'a -> 'b t) : 'b t = match m with
    | Err -> Err
    | Val v -> f v
```

- La monade d'état est plus complexe : on récupère et propage l'état

```
type 'a t = state -> 'a * state
let bind (m : 'a t) (f : 'a -> 'b t) : 'b t =
    fun s ->
        let (a, s) = m s in
        let (b, s) = f a s in
        (b, s)
```

# L'interpréteur monadique I

```
#use "syntax.ml";;
#use "ops.ml";;
#use "state.ml";;
```

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

```
module IdMonad (* : Monad *) = struct
  type 'a t = 'a
  let return v = v
  let bind m f = f m
  let reveal v = v
end
```

## L'interpréteur monadique II

```

module ErrMonad (* : Monad *) = struct
  type 'a t = Err | Val of 'a
  let return v = Val v
  let bind m f = match m with Err → Err | Val v → f v
  let reveal m = match m with
    | Err → assert false
    | Val v → v
end

```

```

module StateMonad (* : Monad *) = struct
  type state = State.t
  type 'a t = state → 'a * state
  let return v = fun s → (v, s)
  let bind m f s = let (v, s) = m s in f v s
  let reveal m = fst (m State.init)
end

```



## L'interpréteur monadique III

```

module InterpGen (M: Monad) =
struct
  let ( >>= ) = M.bind

  let interp_gen interp e env =
    match e with
    | `Var id → M.return (List.assoc id env)
    | `Int i  → M.return (`VInt i)
    | `Bool b → M.return (`VBool b)
    | `Op(o,e1,e2) →
      interp e1 env >>= fun v1 →
      interp e2 env >>= fun v2 →
      M.return (app_op o v1 v2)
    | `If(b,e1,e2) →
      (interp b env >>= function
        | `VBool true → interp e1 env

```

## L'interpréteur monadique IV

```

        | `VBool false -> interp e2 env
        | _ -> failwith "Non_␣boolean_␣test")
| `App (e1,e2) ->
    (interp e1 env >>= function
        | `VFun f -> interp e2 env >>= f
        | _ -> failwith "Application_␣of_␣non_␣function")
| `Fun (id,e) ->
    M.return (`VFun (fun a -> interp e ((id,a)::env)))
| _ -> assert false (* pour permettre des extensions *)
end
    
```

```

module InterpPlain = struct
  type value =
    [ `VInt of int | `VBool of bool
    | `VFun of value -> result ]
  and result = value
  type env = (id * value) list
    
```

## L'interpréteur monadique V

```

include InterpGen(IdMonad)
let rec interp (e:exp) (env : env) : result =
    interp_gen interp e env
end

module InterpErr = struct
  type value =
    [ `VInt of int | `VBool of bool
    | `VFun of value -> result ]
  and result = value ErrMonad.t
  type env = (id * value) list
  include InterpGen(ErrMonad)
  let rec interp exp (env : env) : result = match exp with
    | `Fail -> ErrMonad.Err
    | _ -> interp_gen interp exp env
end

```

## L'interpréteur monadique VI

```

module InterpState = struct
  type value =
    [ `VInt of int | `VBool of bool
      | `VFun of value -> result
      | `VUnit ]
  and result = value StateMonad.t
  type env = (id * value) list
  include InterpGen(StateMonad)
  let to_int = function
    | `VInt i -> StateMonad.return i
    | _ -> failwith "Not an integer"
  let rec interp exp (env : env) : result = match exp with
    | `Unit -> StateMonad.return `VUnit
    | `Seq (e1,e2) ->
        interp e1 env >>= fun _ -> interp e2 env
    | `While (b,e) ->
        interp (`If(b,`Seq(e,exp),`Unit)) env
    
```

## L'interpréteur monadique VII

```

| `Get e ->
  interp e env >>= to_int >>= fun i ->
    fun s -> `VInt (State.get s i), s
| `Set (e1,e2) ->
  interp e1 env >>= to_int >>= fun i ->
  interp e2 env >>= to_int >>= fun x ->
    fun s -> `VUnit, State.set s i x
| _ -> interp_gen interp exp env
end

let prog =
  `App(`Fun("x",`If(`Var "x",`Int 1,`Int 2)),`Bool true)

let _ = InterpPlain.interp prog []
let _ = ErrMonad.reveal (InterpErr.interp prog [])
let _ = StateMonad.reveal (InterpState.interp prog [])
    
```

## Rappel sur les équations

- Nous avons vu les trois axiomes des monades :

$$\begin{aligned} \text{bind } m \text{ return} &= m \\ \text{bind } (\text{return } e) f &= f e \\ \text{bind } (\text{bind } m g) h &= \text{bind } m (\text{fun } x \rightarrow \text{bind } (g x) h) \end{aligned}$$

- En utilisant la notation  $\gg=$  pour `bind` :

$$\begin{aligned} m \gg= \text{return} &= m \\ \text{return } e \gg= f &= f e \\ m \gg= g \gg= h &= m \gg= (\text{fun } x \rightarrow g x \gg= h) \end{aligned}$$

## Utilisation des équations monadiques

- ▶ Les équations monadiques nous permettent de prouver une fois pour toutes des propriétés pour tous les interpréteurs.
- ▶ Par exemple, on peut montrer que l'addition est associative.
 
$$\text{interp}(\text{'Op(Plus,a,'Op(Plus,b,c))}) \text{ env} = \text{interp}(\text{'Op(Plus,'Op(Plus,a,b),c)}) \text{ env}$$
- ▶ Et cela, indépendamment de la monade utilisée pour construire l'interpréteur !

## La preuve d'associativité de l'addition

- On utilise le fait que :

$$\begin{aligned} \text{interp}(\text{Op}(o, e1, e2)) &= \text{interp } e1 \text{ env} \gg= \text{fun } v1 \rightarrow \\ &\quad \text{interp } e2 \text{ env} \gg= \text{fun } v2 \rightarrow \\ &\quad \text{return (app\_op } o \text{ } v1 \text{ } v2) \end{aligned}$$

- Plus l'associativité de l'addition sur les entiers, et la deuxième et troisième équation des monades.



## Conclusions

Au final :

- ▶ Il y a aussi des “motifs” de programmation dans les langages fonctionnels, qui sont parfois difficiles à identifier ; par exemple pour cause d’imbrication.
- ▶ Les motifs identifiés par les monades sont puissants : on peut les utiliser pour coder des interprètes modulaires, ou encore de la compréhension de liste (voir en TP).
- ▶ Les équations des monades permettent d’établir des propriétés générales.

## Quelques repères historiques

Les *monades* sont un concept mathématique apparu dans la théorie des catégories, et qui a été largement repris en informatique :

- 1989 Eugenio Moggi les utilise pour construire une sémantique dénotationnelle modulaire des langages de programmation
- 1992 Philip Wadler popularise l'utilisation des monades en programmation fonctionnelle ; aujourd'hui les monades sont un des concepts les plus utilisés dans la communauté Haskell
- 1995 Peter Buneman, Val Tannen et leurs étudiants construisent des langages de requête concis et optimisables basés sur les monades des collections (similaire à la compréhension des listes)

## Pour en savoir plus



Eugenio Moggi.

Notions of computation and monads.

Inf. Comput., 93(1) :55–92, July 1991.



Philip Wadler.

The essence of functional programming.

In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.



Nick Benton, John Hughes, and Eugenio Moggi.

Monads and effects.

In Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures, pages 42–122, London, UK, UK, 2002. Springer-Verlag.