Programmation Fonctionnelle Avancée 11 : Monades

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

PARIS DIDEROT

treinen@irif.fr

28 novembre 2018

Qu'est-ce que c'est, les Monades ?

- À la base, une monade est un *type polymorpe* `'a t`, qu'on considère comme un *enrichissment* d'un type `'a`.
- Par exemple : passer d'un type `'a` vers un type de listes sur `'a` (la monade des listes, voir le cours 12), ou vers un type produit de `'a` avec quelque chose (voir ce cours pour un exemple).
- Puis, il faut un moyen pour lever les fonctions sur `'a` vers `'a 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 originaires 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
  | `Iden of id
  | `App of (exp * exp)
  | `Abs of (id * exp)
  | `If of (exp * exp * exp)
  | `Comp of (compop * exp * exp)
  | `Base of (op * exp * exp)]
and op      = Plus | Minus | Mult | Div
and compop  = Eq | Less | More;;
```


Exemples (valeurs.ml)

```
(* The type of values. Values can be bound to  *)
(* identifiers.                                *)
type value =
  [
  | `VInt of int
  | `VBool of bool
  | `VArrow of (value -> value)
  ];;

(* The type of results. Results are obtained by *)
(* applying a function or operator to values.
*)
type result = value
```


Exemples (ops1.ml)

```
(* appliquer un operateur arithmetique *)
let appop op v v' =
  match v with
    |`VInt x ->
    (match v' with
     | `VInt y ->
        (match op with
         | Plus -> `VInt (x+y)
         | Minus -> `VInt(x-y)
         | Mult ->  `VInt(x*y)
         | Div ->  `VInt(x/y))
     | _ -> failwith "Non integer in binop")
  | _ -> failwith "Non integer in binop";;
```


Exemples (ops2.ml)

```
(* appliquer un operateur de comparaison *)
let appcompop op v v' =
  match v with
    |`VInt x ->
    (match v' with
     | `VInt y ->
        (match op with
         | Eq -> `VBool (x=y)
         | Less -> `VBool (x<y)
         | More -> `VBool (x>y))
     | _ -> failwith "Non integer in bincomp")
  | _ -> failwith "Non integer in bincomp";;
```

L'interpréteur I

```
let rec var (i,env) =
  match env with
  | []        -> failwith "Unknown variable"
  | (id,a)::r -> if id = i then a else var (i, r);;

let rec interp (exp:exp) env : result =
  match exp with
  | `App (e1,e2) -> let fv = interp e1 env in
      (match fv with
         `VArrow f -> let rv = interp e2 env in f rv
       | _ -> failwith "Application of non function")
  | `Abs (id,exp)-> `VArrow(fun a -> (interp exp ((id,a)::env)))
  | `Iden(id)    -> var(id,env)
  | `Int i           -> `VInt i
  | `Bool b          -> `VBool b
  | `Base(op,e1,e2)  -> appop op (interp e1 env)
                                 (interp e2 env)
```


L'interpréteur II

```
  | `Comp(op,e1,e2)  -> appcompop op (interp e1 env)
                                     (interp e2 env)
  | `If(b,e1,e2) -> let bv = interp b env in
      (match bv with
       | `VBool true -> interp e1 env
       | `VBool false -> interp e2 env
       | _ -> failwith "Non boolean test")
;;

(* Well typed program *)
interp (`App(`Abs("x",`If(`Iden "x",`Int 1,`Int 2)),`Bool true)) [

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


Interpréteur avec erreur

- Voyons maintenant comment on est obligé de modifier le code si on veut adapter l'interpréteur à travailler avec des programmes qui peuvent lever des erreurs.
- 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

```
type value_err = [ `VInt of int | `VBool of bool
                 | `VArrow of (value_err -> result_err) ]
and result_err = Err | Val of value_err;;

let rec interperr exp env : result_err =
  match exp with
  (* the functional core *)
  | `App (e1,e2) -> let fv = interperr e1 env in
      (match fv with Err -> Err
       | Val (`VArrow f) -> let rv = interperr e2 env in
          (match rv with Err -> Err
           | Val v -> f v)
       | _ -> failwith "Application of non function")
  | `Abs (id,exp)-> Val (`VArrow(fun a -> (interperr exp ((id,a)::
  | `Iden(id)    -> Val (var(id,env))
  (* base values and operations *)
  | `Int i           -> Val (`VInt i)
```

L'interpréteur avec gestion d'erreurs II

```
| `Bool b           -> Val (`VBool b)
| `Base(op,e1,e2)  -> let r1 = interperr e1 env in
    (match r1 with Err -> Err
    | Val v1 -> let r2 = interperr e2 env in
        (match r2 with Err -> Err
         | Val v2 -> Val (appop op v1 v2)))
| `Comp(op,e1,e2)  -> let r1 = interperr e1 env in
    (match r1 with Err -> Err
    | Val v1 -> let r2 = interperr e2 env in
       (match r2 with Err -> Err
        | Val v2 -> Val (appcompop op v1 v2)))
(* conditonal *)
| `If(b,e1,e2) -> let bv = interperr b env in
    (match bv with Err -> Err
    | Val (`VBool true) -> interperr e1 env
    | Val (`VBool false) -> interperr e2 env
    | _ -> failwith "Non boolean test")
(* Error *)
```


L'interpréteur avec gestion d'erreurs III

```
  | `Fail -> Err;;

(* Program with no error *)
interperr
 (`App(`Abs("x",`If(`Iden "x",`Int 1,`Int 2)),
       `Bool true)) [];;

(* interperr returns an error value *)
interperr
  (`App(`Abs("x",`If(`Iden "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 actuels de fonctions) du type de *résultats* qui est plus riche.
- Attention, le type des résultats figure dans le type des valeurs (à cause de la fleche fonctionnelle).
- 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 est représenté par une fonction qui associe à l'adresse d'une case mémoire son contenu.
- Le type de résultat change : c'est maintenant une paire d'une valeur et d'un état.
- L'intpréteur doit prendre un argument supplémentaire qui est un état.

L'interpréteur avec état I

```
(* new types of values and results *)
type state = loc -> value_st
and loc = int
and value_st =
  [ `VInt of int | `VBool of bool   (* as before *)
    | `VLoc of loc | `VUnit         (* for imperative *)
    | `VArrow of (value_st -> state -> result_st)
  ]
and result_st = value_st * state;;

(* auxiliary functions needed in the interpreter *)
(* for imperative stuff.                         *)

let emptystate = (fun _ -> failwith "No such cell")

type env = (id * value_st) list;;
```


L'interpréteur avec état II

```
let emptyenv = []

let newlocation = let l = ref 0 in
   fun () -> let nl = !l in l:= nl+1; nl;;

let update (l:loc) v (s:state) =
 let s' = fun l' -> if l = l' then v else s l' in (`VUnit,(s':stat

let lookup (l:loc) (s:state) = (s l),s;;

(* creates a new location, pointing to v. returns a resulting cons
(* of the location, and a new state obtained by addiiton of the ne
*)
(* binding.
*)
let init v (s:state) =
 let (l:loc) = newlocation() in
 let s' = fun l' -> if l = l' then v else s l'
```


L'interpréteur avec état III

```
 in ((`VLoc l:value_st), (s':state));;

let rec interpst exp (env:env) (s:state) : result_st =
  match exp with
  (* functional core *)
    `App (e1,e2) -> let fv = interpst e1 env s in
      (match fv with
       | `VArrow f, s' ->
          let v,s'' = interpst e2 env s' in f v s''
       | _ -> failwith "Non functional value in application")
  | `Abs (id,exp)-> `VArrow(fun a -> (interpst exp ((id,a)::env))), s
  | `Iden(id)    -> var(id,env), s

  (* base types and operations *)
  | `Int i           -> `VInt i, s
  | `Bool b          -> `VBool b, s
  | `Base(op,e1,e2)  -> let v,s' = interpst e1 env s in
                        let v',s'' = interpst e2 env s' in
```


L'interpréteur avec état IV

```
                        appop op v v', s''
  | `Comp(op,e1,e2)  -> let v,s' = interpst e1 env s in
                        let v',s'' = interpst e2 env s' in
                        appcompop op v v', s''

  (* conditional *)
  | `If(b,e1,e2) -> let bv,s' = interpst b env s in
      (match bv with
       | `VBool true -> interpst e1 env s'
       | `VBool false -> interpst e2 env s'
       | _ -> failwith "Non boolean condition in conditional")

  (* statements and references *)
  | `Unit -> `VUnit, s
  | `Seq (e1,e2) -> let _,s' = interpst e1 env s in interpst e2 en
  | `Ref e -> let v,s' = interpst e env s in init v s'
  | `Deref e -> let l,s' = interpst e env s in
                (match l with
```

L'interpréteur avec état V

```
                    | `VLoc loc -> lookup loc s'
                    | _ -> failwith "Not a reference")
  | `Update (e,e') ->
     let l,s' = interpst e env s in
        (match l with
          | `VLoc loc -> let v,s'' = interpst e' env s' in update loc v s
          | _ -> failwith "Not a reference")
;;

(* Program with a sequence and memory access *)
(* # (function x -> if !x then x := false else (); !x) (ref true)
   - : bool = false *)
  interpst
    (`App
      (`Abs("x",
            `Seq(`If(`Deref (`Iden "x"),
                     `Update (`Iden "x",`Bool false),
                     `Unit),
```


L'interpréteur avec état VI

```
                  `Deref (`Iden "x"))),
      `Ref (`Bool true)))
  emptyenv emptystate;;
```


Tout change I

Nous avons du *tout changer* dans le code de notre interpréteur.
Comparons le cas de l'application :

```
(* langage simple *)
    `App (e1,e2) -> let fv = interp e1 env in
      (match fv with
        `VArrow f -> let rv = interp e2 env in
                     f rv

(* avec erreurs *)
    `App (e1,e2) -> let fv = interperr e1 env in
      (match fv with Err -> Err
      | Val(`VArrow f) -> let rv = interperr e2 env in
         (match rv with Err -> Err
          | Val v -> f v)

(* avec etat *)
    `App (e1,e2) -> let fv =  interpst e1 env s in
      (match fv with
        `VArrow f, s' ->
          let v,s'' = interpst e2 env s' in f v s''
```


Tout change II

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

```
(* langage simple *)
  | `Abs (id,exp)-> `VArrow(fun a -> (interp exp ((id,a)::env)))
  | `Iden(id)    -> var(id,env)

(* avec erreurs *)
  | `Abs (id,exp)-> Val (`VArrow(fun a ->
                                 (interperr exp ((id,a)::env))))
  | `Iden(id)    -> Val (var(id,env))

(* avec etat *)
  | `Abs (id,exp)-> `VArrow(fun a ->
                            (interpst exp ((id,a)::env))), s
  | `Iden(id)    -> var(id,env), s
```

C'est désespérant ! Peut-on faire mieux ?

Les monades à la rescousse !

- Il y a à la base l'idée de distinguer un type de valeurs 'a d'un type représentant les *résultats d'un calcul* de type 'a.
- Une *monade* est constituée par
  - un type polymorphe *'a t*
  - une opération ***bind***: *'a t -> ('a -> 'b t) -> 'b t*
  - une opération ***return***: *'a -> 'a t*
  - satisfaisant certaines équations.
- Le type *'a t* contient des *résults de calculs* de type *'a*
- **bind** compose un *résultat de calcul* de type 'a, avec un consommateur d'une valeur de type 'a.
- **return** plonge une valeur dans un résultat de calcul.

Propriétés des opérateurs de monade

Les trois équations suivantes doivent être satisfaites :

1. bind c return = c
2. bind (return v) f = f v
3. bind e1 (fun x -> bind e2 (fun y -> e3)) =
   bind (bind e1 (fun x -> e2)) (fun y -> e3)
   si x non libre dans e3
   C'est une propriété d'associativité de bind.

Extensions

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

- On verra que pour la compréhension des listes il faut aussi une opération constante *zero : 'a t* avec l'équation *bind zero f = zero* pour coder les compréhensions.
- 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

Dans le cas de notre évaluateur, nous remarquons qu'il s'agit d'une fonction de type :

```
interp : exp -> (id * idt) list -> expt
```

où :

- exp est le type des expressions à évaluer
- id est le type des identificateurs
- idt est le type des valeurs qu'on peut associer à un identificateur
- expt est le type des résultats retournées par l'interpréteur

Idt, Expt et return

- Voici la table des types pour les trois cas :

langage	idt (= 'a)	expt (= 'a t)
simple	value	value
erreur	value_err	Err | Val of value_err
etat	value_st	state -> value_st * state

- On peut voir expt comme l'instantiation de trois monades, pour lesquelles il est facile de définir **return** :

monade T	'a t	return$_T$
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 réécrire notre code de façon unifiée en utilisant la fonction *return* ; dans les trois cas, on aura

```
(* langage simple, erreurs, etat *)
| 'Abs (id,exp)-> return
                    ('VArrow(fun a ->
                                (interp exp ((id,a)::env))))
| 'Iden(id)     -> return (var(id,env))
```

- Et cela s'étend naturellement à tous les cas des *valeurs* du langage :

```
(* langage simple, erreurs, etat *)
| 'Int i            -> return ('VInt i)
| 'Bool b           -> return ('VBool b)
```

- Vérifiez le fait qu'en utilisant la monade appropriée on obtient bien le code correspondant à l'interpréteur écrit auparavant.

Trouver le bon bind

Rappelons le code de l'application :

```
(* langage simple *)
    'App (e1,e2) -> let fv = interp e1 env in
      (match fv with
      | 'VArrow f -> let rv = interp e2 env in f rv

(* avec erreurs *)
    'App (e1,e2) -> let fv = interperr e1 env in
      (match fv with Err -> Err
      | Val ('VArrow f) -> let rv = interperr e2 env in
         (match rv with Err -> Err
          | Val v -> f v)

(* avec etat *)
    'App (e1,e2) -> let fv = interpst e1 env s in
      (match fv with
        'VArrow f, s' ->
          let v,s'' = interpst e2 env s' in f v s''
```

Ici, c'est moins facile de repérer l'expansion de **bind**, car il y en a deux imbriquées : une pour e1, et une pour e2 !

Le bon bind

- Voici la bonne définition de **bind** : 'a t -> ('a -> 'b t) -> 'b t pour chaque monade 'a t

monade T	'a t	bind$_T$
identité	'a	fun m f -> f m
erreur	Err | Val of 'a	fun m f -> match m with Err -> Err | Val v -> f v
état	state -> 'a * state	fun m f -> fun s -> let v,s' = m s in f v s'

- On peut maintenant réécrire notre interpréteur complètement, tel que seulement les parties concernant les constructions nouvelles changent !

L'interpréteur monadique I

```
#use "syntax.ml";;
#use "ops1.ml";;
#use "ops2.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 IdM = 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 ErrM = 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 -> failwith "Error"
      | Val v -> v
end;;

module StateM = struct
    type state = int -> int
    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'
```


L'interpréteur monadique III

```
    (* Uninitialised memory *)
    let emptys =
      (fun l -> failwith "Uninitialised memory location")
    let reveal m = let (v,s') = m emptys in v
    (* newloc, update, lkp, init *)
    let newloc = let l = ref 0 in
       fun () -> let nl = !l in l:= nl+1; nl
    let update l v s =
     let s' = fun l' -> if l = l' then v else s l' in ('VUnit,s')
    let lkp l s = (s l),s
    let init v (s:state) =
     let l = newloc() in
     let s' = fun l' -> if l = l' then v else s l' in 'VLoc l,s
end;;

module Interp (M: Monad) =
struct
 type idt = (* les valeurs *)
```


L'interpréteur monadique IV

```
          [  'VInt of int | 'VBool of bool
           | 'VArrow of (idt -> expt) ]
 and  expt = idt M.t (* les expressions *)

 let ( >>= ) = M.bind
 let return = M.return

 let rec var (i,env) =
  match env with
  | []        -> failwith "Unknown variable"
  | (id,a)::r -> if id = i then a else var (i, r);;

 let rec interp (exp:exp) (env: (id * idt) list) : expt =
   match exp with
     'App (e1,e2) -> interp e1 env >>= fun fv ->
          (match fv with
            'VArrow f -> interp e2 env >>= fun rv -> f rv
          | _ -> failwith "Application of non function")
```

L'interpréteur monadique V

```
  | `Abs (id,exp)->
      return
        (`VArrow(fun a -> (interp exp ((id,a)::env))))
  | `Iden(id)     -> return (var(id,env))
  | `Int i        -> return (`VInt i)
  | `Bool b       -> return (`VBool b)
  | `Base(op,e1,e2)  ->
        interp e1 env >>= fun v1 ->
        interp e2 env >>= fun v2 ->
         return (appop op v1 v2)
  | `Comp(op,e1,e2)  ->
        interp e1 env >>= fun v1 ->
        interp e2 env >>= fun v2 ->
         return (appcompop op v1 v2)
  | `If(b,e1,e2) -> interp b env >>= fun bv ->
     (match bv with
        `VBool true -> interp e1 env
      | `VBool false -> interp e2 env
```


L'interpréteur monadique VI

```
        | _ -> failwith "Non boolean test")
end;;

module IntPlain = Interp(IdM);;

module IntErr = Interp(ErrM);;

module IntState = Interp(StateM);;

let prog : exp =
 (`App(`Abs("x",`If(`Iden "x",`Int 1,`Int 2)),`Bool true));;

IntPlain.interp prog [];;
ErrM.reveal (IntErr.interp prog []);;
StateM.reveal (IntState.interp prog []);;
```


Rappel sur les équations

- Nous avons vu les trois axiomes des monades :

  bind m return = m

  bind (return e) f = f e

  bind e1 (fun x -> bind e2 (fun y -> e3)) =
  bind (bind e1(fun x -> e2)) (fun y -> e3)
  si x non libre dans e3

- On écrit souvent l'opération `bind` comme un opérateur infixe $\gg=$.
- Cela rend les axiomes un peu plus lisibles :

  $m \gg= return = m$

  $return\ e \gg= f = f\ e$

  $e1 \gg= (fun\ x \to e2 \gg= fun\ y \to e3) =$
  $(e1 \gg= fun\ x \to e2) \gg= fun\ y \to e3$ *si x non libre dans e3*

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.

  interp (`Base(Plus,e1,`Base(Plus,e2,e3))) env =
  interp (`Base(Plus,`Base(Plus,e1,e2),e3)) env
- ... et cela, indépendamment de la monade utilisée pour construire l'interpréteur !

La preuve d'associativité de l'addition

- On utilise le fait que

```
interp('Base(op,e1,e2))  =  interp e1 env >>= fun v1 ->
                            interp e2 env >>= fun v2 ->
                            return (appop op v1 v2)
```

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

Conclusions

Nous avons appris (avec pas mal de sueur) que :

- Il y a des "motifs" de programmation dans les langages fonctionnels aussi, mais qui sont plus difficiles à identifier ; cela vient du fait qu'on les retrouve des fois imbriqués, comme dans le cas de l'application dans l'interprète.
- Les motifs identifiés par les monades sont puissants : on peut les utiliser pour coder la comprehension des listes, ou pour écrire des interprètes modulaires, comme dans notre cas.
- 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 ce qu'on verra avec la comprehension 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.