Programmer avec des monades

- Introduire la programmation monadique
- en donnant des exemples ou cela apporte beaucoup

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 a vu avec la comprehension des listes)

Faire évoluer un évaluateur

Ce n'est pas si simple ...

Ecrire 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.
Pour l'évaluateur complet, voir le code associé aux notes de cours.

La syntaxe du langage

```
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;;

type res =
 [ `Int of int | `Bool of bool
 | `Arrow of (res -> res) ];;
```


Les opérations de base I

```
let getop op v v' =
  match v with `Int x ->
    (match v' with `Int y ->
      (match op with
        Plus -> `Int (x+y)
      | Minus -> `Int(x-y)
      | Mult ->  `Int(x*y)
      | Div ->  `Int(x/y))
    | _ -> failwith "Non integer in binop")
  | _ -> failwith "Non integer in binop";;
val getop : op -> [> `Int of int ] -> [> `Int of int ]
  ] =
  <fun>
```


Les opérations de base II

```
let getcompop op v v' =
  match v with `Int x ->
    (match v' with `Int y ->
      (match op with
        Eq -> `Bool (x=y)
      | Less -> `Bool (x<y)
      | More -> `Bool (x>y))
    | _ -> failwith "Non integer in bincomp")
  | _ -> failwith "Non integer in bincomp";;
val getcompop :
  compop -> [> `Int of 'a ] -> [> `Int of 'a ] -> [> `Bool
fun>
```


L'environnement

Pour executer un programme qui contient des variables, l'interprète utilise un *environnement*, qui associe à chaque variable sa valeur.
La fonction suivant retrouve la valeur d'une variable dans un environnement.

```
let rec var (i,env) =
  match env with
    []        -> failwith "Unknwown variable"
  | (id,a)::r -> if id = i then a else var (i, r);;
val var : 'a * ('a * 'b) list -> 'b = <fun>
```

L'interpéte I

```
let rec interp (exp:exp) env : res=
  match exp with
    'App (e1,e2) -> let fv = interp e1 env in
      (match fv with
        'Arrow f -> let rv = interp e2 env in f rv
      | _ -> failwith "Application of non function")
  | 'Abs (id,exp)-> 'Arrow(fun a -> (interp exp ((id,a)::env)))
  | 'Iden(id)    -> var(id,env)
  | 'INT i           -> 'Int i
  | 'BOOL b          -> 'Bool b
  | 'Base(op,e1,e2)  -> getop op (interp e1 env) (interp e2 env)
  | 'Comp(op,e1,e2)  -> getcompop op (interp e1 env) (interp e2 env)
  | 'IF(b,e1,e2) -> let bv = interp b env in
      (match bv with
        'Bool true -> interp e1 env
      | 'Bool false -> interp e2 env
      | _ -> failwith "Non boolean test")
;;
val interp : exp -> (id * res) list -> res = <fun>
```


L'interpéte II

```
(* Well typed program *)
interp ('App('Abs("x",'IF('Iden "x",'INT 1,'INT 2)),'BOOL true)) [];;
- : res = 'Int 1


(* Ill typed program *)
interp ('App('INT 1, 'INT 2))[];;
Exception: Failure "Application of non function".
```


Interprète avec erreur

Voyons maintenant comment on est obligés de modifier le code si on veut adapter l'interprète 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:

```
type resok = [ 'Int of int | 'Bool of bool
             | 'Arrow of (resok -> reserr) ]
and reserr = Err | Res of resok;;
```

Attention au cas de la fonction: nous écrivons un interprète en *appel par valeur*, donc le paramètre n'est jamais un erreur, parce-que l'erreur aurait été capturé avant, lors de l'évaluation de l'argument.

Interprète avec erreur I

Chaque fois qu'on *utilise* le résultat d'un appel à l'interprèteur, on doit tester s'il s'agit d'un erreur, et le propager.
Le code de l'intepréteur s'alourdit considérablement:

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

Interprète avec erreur II

```
  | `Base(op,e1,e2)  -> let r1 = interperr e1 env in
      (match r1 with Err -> Err
      |  Res v1 -> let r2 = interperr e2 env in
          (match r2 with Err -> Err
           | Res v2 -> Res (getop op v1 v2)))
  | `Comp(op,e1,e2)  -> let r1 = interperr e1 env in
      (match r1 with Err -> Err
      |  Res v1 -> let r2 = interperr e2 env in
         (match r2 with Err -> Err
           | Res v2 -> Res (getcompop op v1 v2)))
  (* conditonal *)
  | `IF(b,e1,e2) -> let bv = interperr b env in
      (match bv with Err -> Err
      | Res (`Bool true) -> interperr e1 env
      | Res (`Bool false) -> interperr e2 env
      | _ -> failwith "Non boolean test")
  (* Error *)
  | `Fail -> Err;;
val interperr :
  ([< `Abs of 'b * 'a
```


Interprète avec erreur III

```
||      | `App of 'a * 'a
||      | `BOOL of bool
||      | `Base of op * 'a * 'a
||      | `Comp of compop * 'a * 'a
||      | `Fail
||      | `IF of 'a * 'a * 'a
||      | `INT of int
||      | `Iden of 'b ]
||    as 'a) ->
||   ('b * resok) list -> reserr = <fun>


(* Program with no error *)
interperr
 (`App(`Abs("x",`IF(`Iden "x",`INT 1,`INT 2)),`BOOL true)) [];;
|| - : reserr = Res (`Int 1)
```


Interprète avec erreur IV

```
(* Program with an error *)
let errp =
 `App(
   `Abs("x",`IF(`Iden "x",`Fail,`INT 2)),
   `BOOL true);;

(* interperr returns an error value *)
interperr errp [];;
|| - : reserr = Err
```


Ajout d'un ètat

Ajoutons maintenant un état à notre interprète: il est représenté par une fonction qui associe à l'adresse d'une case mémoire son contenu.

```
type state = loc -> stres
and loc = int
and stres =
 [ `Int of int | `Bool of bool | `Loc of loc
 | `Arrow of (stres -> state -> stres * state) | `Unit
 ];;


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

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

Modifier l'état

On peut créer, lire et modifier les cases mémoire avec les fonctions suivantes:

```
let newloc = let l = ref 0 in
   fun () -> let nl = !l in l:= nl+1; nl;;
|| val newloc : unit -> int = <fun>

let update (l:loc) v (s:state) =
 let s' = fun l' -> if l = l' then v else s l' in ('Unit,(s':state));;
|| val update : loc -> stres -> state -> [> 'Unit ] * state = <fun>

let lkp (l:loc) (s:state) = (s l),s;;
|| val lkp : loc -> state -> stres * state = <fun>

let init v (s:state) =
 let (l:loc) = newloc() in
 let s' = fun l' -> if l = l' then v else s l'
 in (('Loc l:stres), (s':state));;
|| val init : stres -> state -> stres * state = <fun>
```


Interprète avec ètat I

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

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


Interprète avec ètat II

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

  (* statements and references *)
  | 'UNIT -> 'Unit, s
  | 'Seq (e1,e2) -> let _,s' = interpst e1 env s in interpst e2 env s'
  | '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 'Loc loc -> lkp loc s'
                 | _ -> failwith "Not a reference")
  | 'Update (e,e') ->
     let l,s' = interpst e env s in
        (match l with
          'Loc loc -> let v,s'' = interpst e' env s' in update loc v s'
        | _ -> failwith "Not a reference")
;;
```


Interprète avec ètat III

```
|| val interpst :
||   ([< 'Abs of id * 'a
||     | 'App of 'a * 'a
||     | 'BOOL of bool
||     | 'Base of op * 'a * 'a
||     | 'Comp of compop * 'a * 'a
||     | 'Deref of 'a
||     | 'IF of 'a * 'a * 'a
||     | 'INT of int
||     | 'Iden of id
||     | 'Ref of 'a
||     | 'Seq of 'a * 'a
||     | 'UNIT
||     | 'Update of 'a * 'a ]
||    as 'a) ->
||   env -> state -> stres * state = <fun>
```

Interprète avec ètat IV

```
(* Program with a sequence and memory access *)
(* # let x = ref true in if !x = true then x := false else (); !x
   - : bool = false *)
interpst
 (`App
   (`Abs("x",
    `Seq(`IF(`Deref (`Iden "x"),
             `Update (`Iden "x",`BOOL false),
            `UNIT),
          `Deref (`Iden "x"))),
   `Ref (`BOOL true)))
 [] emptyst;;
|| *                  - : stres * state = (`Bool false, <fun>)
```


Bilan des courses

On ne va pas loin comme ça!

Tout change I

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

```
(* langage simple *)
    `App (e1,e2) -> let fv = interp e1 env in
      (match fv with
        `Arrow 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
      | `Arrow f -> let rv = interperr e2 env in
         (match rv with Err -> Err
          | v -> f v)

(* avec etat *)
    `App (e1,e2) -> let fv =  interpst e1 env s in
      (match fv with
        `Arrow 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)-> `Arrow(fun a -> (interp exp ((id,a)::env)))
  | `Iden(id)    -> var(id,env)

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

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

C'est desespérant! Peux-t-on faire mieux?

Les monades à la rescousse

Structurer les valeurs de l'interprète

Définition

Une *monade* est constitué par

- un type polymorphe *'a t*
- une opération ***bind***: *'a t -> ('a -> 'b t) -> 'b t*
- une opération ***return***: *'a -> 'a t*

et satisfaisant les trois équations vues lors du dernier cours pour les listes:

```
                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
```

Ces équations sont utiles pour prouver des propriétés, mais elles sont souvent oubliées par les programmeurs...

L'intuition

Le type *'a t* contient des *calculs* sur des valeurs de type *'a*, et les deux opérations ***bind***: *'a t -> ('a -> 'b t) -> 'b t* et ***return***: *'a -> 'a t* fournissent les briques de base pour construire et composer des calculs.

Une monade particulière peut fournir d'autres briques pour construire des calculs:

- dans le cours sur la comprehension des listes, par exemple, on a vu qu'il nous fallait 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 resultats retournées par l'interprète

Idt, Expt et return

Voici la table des types pour les trois cas:

langage	idt	expt
simple	res	res
erreur	res	Err | Res of res
état	stres	state -> stres * state

On peut voir expt comme l'instantiation de trois modades, pour lequelles il est facile de définir **return** :

monade T	'a t	return$_T$
identité	'a	fun v -> v
erreur	Err | Res of 'a	fun v -> Res v
état	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
          ('Arrow(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 ('Int i)
  | 'BOOL b             -> return ('Bool b)
```

Vérifiez par vous mêmes le fait qu'en utilisant la monade appropriée on obtient bien le code correspondant à l'interprète 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
         'Arrow 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
      | Res ('Arrow f) -> let rv = interperr e2 env in
          (match rv with Err -> Err
           | Res v -> f v)

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

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

Le bon bind

Voici la bonne definition 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 | Res of 'a	fun m f -> match m with Err -> Err | Res 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 interpète complétement, de façon que ne changent que les parties concernant les constructions nouvelles!

Interpreteur monadique I

```
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;;
```


Interpreteur monadique II

```
module ErrM = struct
   type 'a t = Err | Res of 'a
   let return v = Res v
   let bind m f =
     match m with Err -> Err | Res v -> f v
   let raise () = Err
   let reveal m =
     match m with
           Err -> failwith "Error"
         | Res v -> v
 end;;
```


Interpreteur monadique III

```
module StateM = struct
    type s = int -> int
    type 'a t = s -> 'a * s
    let return v = fun s -> (v,s)
    let bind m f s =
       let (v,s') = m s in f v s'
    (* 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 ('Unit,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 'Loc l,s
 end;;
```


Interpreteur monadique IV

```
module Interp (M: Monad) =
struct
 type idt = (* les valeurs *)
          [  'Int of int | 'Bool of bool
           | 'Arrow of (idt -> expt) ]
 and  expt = idt M.t (* les expressions *)

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

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

Interpreteur monadique V

```
    | `INT i       -> return (`Int i)
    | `BOOL b      -> return (`Bool b)
    | `Base(op,e1,e2)  ->
          interp e1 env >>= fun v1 ->
          interp e2 env >>= fun v2 ->
           return (getop op v1 v2)
    | `Comp(op,e1,e2)  ->
          interp e1 env >>= fun v1 ->
          interp e2 env >>= fun v2 ->
           return (getcompop op v1 v2)
    | `IF(b,e1,e2) -> interp b env >>= fun bv ->
       (match bv with
          `Bool true -> interp e1 env
        | `Bool false -> interp e2 env
        | _ -> failwith "Non␣boolean␣test")
end;;
```


Interpreteur monadique VI

```
module IntPlain = Interp(IdM);;
|| module IntPlain :
||   sig
||     type idt = [ `Arrow of idt -> expt | `Bool of bool | `Int of int ]
||     and expt = idt IdM.t
||     val ( >>= ) : 'a IdM.t -> ('a -> 'b IdM.t) -> 'b IdM.t
||     val return : 'a -> 'a IdM.t
||     val interp : exp -> (id * idt) list -> expt
||   end


module IntErr = Interp(ErrM);;
|| module IntErr :
||   sig
||     type idt = [ `Arrow of idt -> expt | `Bool of bool | `Int of int ]
||     and expt = idt ErrM.t
||     val ( >>= ) : 'a ErrM.t -> ('a -> 'b ErrM.t) -> 'b ErrM.t
||     val return : 'a -> 'a ErrM.t
||     val interp : exp -> (id * idt) list -> expt
||   end
```


Interpreteur monadique VII

```
module IntState = Interp(StateM);;
|| module IntState :
||   sig
||     type idt = [ `Arrow of idt -> expt | `Bool of bool | `Int of int ]
||     and expt = idt StateM.t
||     val ( >>= ) : 'a StateM.t -> ('a -> 'b StateM.t) -> 'b StateM.t
||     val return : 'a -> 'a StateM.t
||     val interp : exp -> (id * idt) list -> expt
||   end
```


Interpreteur monadique VIII

```
let prog : exp =
 (`App(`Abs("x",`IF(`Iden "x",`INT 1,`INT 2)),`BOOL true));;
|| val prog : exp =
||   `App (`Abs ("x", `IF (`Iden "x", `INT 1, `INT 2)), `BOOL true)


IntPlain.interp prog [];;
|| - : IntPlain.expt = `Int 1

ErrM.reveal (IntErr.interp prog []);;
|| - : IntErr.idt = `Int 1

StateM.reveal (IntState.interp prog []);;
|| - : IntState.idt = `Int 1
```

Vérifiez par vous mêmes le fait qu'un utilisant la monade appropriée on obtient bien le code correspondant à l'interprète écrit auparavant.

Prouver des propriétés génériques

il suffit que **bind** et **return** viennent bien d'une monade...

Rappel sur les équations

Les équations que nous avons vu au début

```
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
```

S'écrivent comme suit avec l'opérateur infixe

```
m >>= return                           =  m
return e >>= f                         =  f e
e1 >>= fun x -> e2 >>= fun y -> e3     =
(e1 >>= fun x -> e2) >>= fun y -> e3       si x non libre dans e3
```


Utilisation des equations monadiques

Les équations monadiques nous permettent de prouver une fois pour toutes des propriétes pour tous les interprètes.

Par exemple, on peut montrer que l'addition est associative

```
intep (`Base(Plus,e1,`Base(Plus,e2,e3)))  env   =
intep (`Base(Plus,`Base(Plus,e1,e2),e3))  env
```

et cela, independamment de la monade utilisée pour construire l'interprète!

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 (getop op v1 v2)
```

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

Voir la preuve faite au tableau

Combiner des monades avec les *transformateurs de monades*

Modulariser les monades!

Comment avoir état et erreur ensemble?

Une fois que nous savons écrire un interprète modularisé grâce aux monades, on peut vouloir *combiner* différentes monades ensemble.

Par exemple, pour un langage avec des erreurs *et* des effets de bord.

On peut essayer de trouver les bons **return** et **bind** pour ce cas... mais ce n'est pas immédiat:

- d'abord, en notant 'a maybe = Err | Res **of** 'a, est-ce que nous voulons

  $$\text{'a t} = \text{state} \rightarrow (\text{'a} * \text{state})\ \text{maybe}$$

  qui correspond à des transactions, ou

  $$\text{'a t} = \text{state} \rightarrow \text{'a maybe} * \text{state}$$

  qui correspond à des langages traditionnels?
- et pour chaque combinaison, ècrire **bind** peut être fatiguant...

Les transformateurs de monades

Les mathématiciens savent que, en gènèral, les monades ne se composent pas, mais Eugenio Moggi a indiqué comment s'en sortir en utilisant des *transformateurs de monades*.

En OCaml, il s'agit de foncteurs qui prennent une monade M en entrée, et produisent en sortie une monade M' à laquelle on a ajouté des fonctionnalités supplémentaires (les erreurs, les états, etc.).

Ces foncteurs fournissent aussi une fonction
lift : 'a M.t -> 'a M'.t qui permet de transformer les calculs dans la monade M en calculs dans M'.

Nous allons voir sur le code fourni à part comment écrire ces transformateurs pour létat et les erreurs, et comment s'en servir.

Conclusions

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

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.