Programmation Avancée

Effets et Monades

Armaël Guéneau

Monades

Un mot compliqué pour un concept pas si compliqué

Monade : une *abstraction* pour une notion de *calcul*

En programmation fonctionnelle, permet de *structurer du code*

Permet d'écrire des programmes *plus modulaires*

Programmation monadique

Une monade : une notion de *calcul fournissant certaines opérations*

Permet de structurer des programmes comme la composition de :

- Implémentation de la monade (*réutilisable*) et de ses opérations
- Programme utilisant la monade (“*code monadique*”)

Antidote au code écrit façon “grosse boule de boue” où tout est fait en même temps

Exemple

Exemple : variations pour calculatrice

$\rightarrow$ voir `calc.ml`

calculatrice : observations

Dans `calc.ml`, on observe :

Chaque monade permet de séparer la logique du code (la fonction `eval` de la calculatrice) des **opérations primitives** sur lesquelles elle s'appuie.

une notion de calcul

Pour chaque variante de la calculatrice, on a utilisé :

- un type `'a t` des **calculs** renvoyant des valeurs de type `'a`
- `return` pour injecter des valeurs de `'a` dans `'a t`
- `>>=` pour enchainer des calculs

ainsi que les opérations spécifiques à chaque implémentation de `'a t`

monade : définition

Une monade est donnée par un type paramétré `'a t`, et deux opérations :

- `return : 'a -> 'a t`
- `(>>=) : 'a t -> ('a -> 'b t) -> 'b t`

(`>>=` est appelé “bind”)

Par ailleurs, `return` et `>>=` doivent satisfaire certaines **lois monadiques** (ce ne peut pas être des fonctions arbitraires).

monades sur l'étagère (à savoir utiliser et implémenter !)

On a vu :

- monade d'erreur
- monade de lecture depuis un environnement
- monade de comptage de "ticks"
- monade d'état aléatoire

monades sur l'étagère (à savoir utiliser et implémenter !)

On a vu :

- monade d'erreur
- monade de lecture depuis un environnement
  $\rightarrow$ **aussi appelée monade de lecture (`Reader`)**
- monade de comptage de "ticks"
- monade d'état aléatoire

monades sur l'étagère (à savoir utiliser et implémenter !)

On a vu :

- monade d'erreur
- monade de lecture depuis un environnement
  → **aussi appelée monade de lecture (`Reader`)**
- monade de comptage de "ticks"
  → **cas particulier de monade d'écriture (`Writer`)**
- monade d'état aléatoire

monades sur l'étagère (à savoir utiliser et implémenter !)

On a vu :

- monade d'erreur
- monade de lecture depuis un environnement
  → **aussi appelée monade de lecture (`Reader`)**
- monade de comptage de "ticks"
  → **cas particulier de monade d'écriture (`Writer`)**
- monade d'état aléatoire
  → **cas particulier de monade d'état (`State`)**

monade `Reader` : calculs ayant accès à un environnement global

```
module type ENVTY = sig type t end
module ReadM (Env : ENVTY) : sig
  type 'a t
  (* return, >>= *)
  val read : Env.t t
  val run : 'a t -> Env.t -> 'a
end
```

(on peut aussi ajouter une opération pour localement modifier l'environnement pendant un sous-calcul)

monade `Writer` : calculs produisant des données en sortie

```
module type OUTPUT = sig
  type t
  val empty : t
  val append : t -> t -> t
end
module WriteM (Out : OUTPUT) : sig
  type 'a t
  (* return, >>= *)
  val write : Out.t -> unit t
  val run : 'a t -> 'a * Out.t
end
```

(on peut aussi ajouter une opération permettant de récupérer la sortie d'un calcul intermédiaire)

monade `State` : calculs avec état mutable global

```
module type STATE = sig type t end

module StateM (St : STATE) : sig
  type 'a t
  (* return, >>= *)
  val get : St.t t
  val put : St.t -> unit t
  val run : 'a t -> St.t -> 'a * St.t
end
```

Intuition : calculs manipulant *une* référence globale stockant une valeur de type `St.t`.

Implémenter des monades

Règles du jeu

Le principe est de fournir un type `'a t` représentant une notion de « calcul » équipé de certaines opérations

en l'implémentant en utilisant du code **purement fonctionnel**

(99% du temps)

monade d'erreur

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val fail : 'a t
```

Implémentation ?

monade d'erreur

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val fail : 'a t
```

Implémentation ? Intuition : `'a t` = calcul de `'a` qui a pu échouer

monade d'erreur

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val fail : 'a t
```

Implémentation ? Intuition : 'a t = calcul de 'a qui a pu échouer

```
type 'a t = 'a option
let return x = Some x
let fail = None
let (>>=) m f =
  match m with
  | None -> None
  | Some x -> f x
```

remarque

l'implémentation n'est pas compliquée !

la puissance des monades est d'identifier ce **concept très général** de « calcul que l'on peut séquencer »

l'implémentation d'une monade n'a pas besoin d'être compliquée pour être utile pour structurer du code

monade de lecture

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val read : Env.t t
```

Implémentation ?

monade de lecture

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val read : Env.t t
```

Implémentation ?

Intuition : `'a t` = calcul qui reçoit un `Env.t` et produit `'a`

monade de lecture

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val read : Env.t t
```

Implémentation ?

Intuition : `'a t` = calcul qui reçoit un `Env.t` et produit `'a`

```
type 'a t = Env.t -> 'a
let return x = fun env -> x
let read = fun env -> env
let (>>=) m f = fun env ->
  f (m env) env
```

monade d'état

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val get : St.t t
val put : St.t -> unit t
```

Implémentation ?

monade d'état

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val get : St.t t
val put : St.t -> unit t
```

Implémentation ?

Intuition : `'a t` = calcul qui reçoit le `St.t` courant et renvoie un `'a` et le nouveau `St.t`

monade d'état

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val get : St.t t
val put : St.t -> unit t
```

Implémentation ?

Intuition : `'a t` = calcul qui reçoit le `St.t` courant et renvoie un `'a` et le nouveau `St.t`

```
type 'a t = St.t -> 'a * St.t
let return x = fun s -> (x, s)
let get = fun s -> (s, s)
let put s = fun _ -> ((), s)
let (>>=) m f = fun s ->
  let (x, s') = m s in
  f x s'
```

lois monadiques

l'implémentation d'une monade doit satisfaire des **lois monadiques**

celles-ci spécifient les propriétés de compatibilité requises entre `return` et `(>>=)`

lois monadiques (1)

```
(return v) >>= fun x -> m
=
let x = v in m
```

Exemple (monade d'erreur, `type 'a t = 'a option`) :

```
match Some v with
| None -> None
| Some x -> m
=
let x = v in m
```

lois monadiques (2)

```
m >>= fun x -> return x
=
m
```

Exemple (monade d'erreur) :

```
match m with
| None -> None
| Some x -> Some x
=
m
```

lois monadiques (3)

```
(mx >>= fun x -> my) >>= fun y -> m
=
mx >>= (fun x -> (my >>= fun y -> m))
```

Exemple (monade d'erreur) :

```
match (match mx with
       | None -> None
       | Some x -> my) with
| None -> None
| Some y -> m
=
match mx with
| None -> None
| Some x ->
  match my with
  | None -> None
  | Some y -> m
```

Plus de monades

autres exemples de monades :

- non-déterminisme
- concurrence (la bibliothèque OCaml `Lwt` fournit une monade !)
- parsing (en TP)

autres exemples de monades :

- **non-déterminisme**
- concurrence (la bibliothèque OCaml `Lwt` fournit une monade !)
- parsing (en TP)

monade de non-déterminisme

`'a t` : un calcul qui produit zéro, une ou plusieurs valeurs de type `'a`

Très pratique pour exprimer facilement des algorithmes de recherche de solution/de preuve.

On peut les exprimer comme des programmes *monadiques* utilisant la monade de non-déterminisme.

$\rightarrow$ voir `nondet.ml`

Monades vs Effets

Il y a un lien très fort entre la notion de *monade*
et celle d'≪ *effet de bord* ≫.

OCaml : un langage avec “effets de bord”

- que sont des effets de bord ?
- qu’ont-ils en commun ?
- existe t-il d’autres effets de bord pas disponibles en OCaml ?

Monades :

- une approche générale pour programmer avec des opérations “custom” (définies par l’utilisateur)
- un moyen de donner une sémantique aux effets

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

- **état mutable (d’ordre supérieur) :**

```
r := f; !r ()
```

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

- **état mutable (d’ordre supérieur) :**

```
r := f; !r ()
```

- **exceptions :**

```
raise Not_found
```

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

- **état mutable (d’ordre supérieur) :**

```
r := f; !r ()
```

- **exceptions :**

```
raise Not_found
```

- **entrées sorties :**

```
print_int 42; read_int ()
```

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

- **état mutable (d'ordre supérieur) :**

```
r := f; !r ()
```

- **exceptions :**

```
raise Not_found
```

- **entrées sorties :**

```
print_int 42; read_int ()
```

- **concurrence :**

```
Thread.create f x, Gc.finalise f v
```

OCaml : effets de bord primitifs

Effets “primitifs” disponibles en OCaml :

- **état mutable (d'ordre supérieur) :**

```
r := f; !r ()
```

- **exceptions :**

```
raise Not_found
```

- **entrées sorties :**

```
print_int 42; read_int ()
```

- **concurrence :**

```
Thread.create f x, Gc.finalise f v
```

- **non-terminaison :**

```
let rec f x = f x
```

Qu'est-ce qu'un effet ?

Qu'est-ce qu'un effet ?

Un **effet** : tout ce que fait une fonction à part calculer un résultat en fonction de ses arguments

Si une expression `E` s'évalue en une valeur `V`, et changer :

```
let x = E
  in E'
```

en

```
let x = V
  in E'
```

change le comportement du programme, alors `E` utilise des effets.

autres effets

Effets primitifs non disponibles en OCaml

- **non-déterminisme**
- **continuations de première classe**
- **effets algébriques et handlers** (introduits en OCaml 5.0)

mais aussi : pas de typage des effets primitifs en OCaml

vision 1 : effets comme primitives

L'état mutable, les exceptions, etc sont ici des effets *primitifs* à OCaml

Vision 1 : les effets correspondent à des *primitives* du langage qui ont des "comportements spéciaux", et changent donc la manière dont on peut raisonner sur les programmes

vision 2 : effets pour structurer les programmes

Vision 2 : les effets sont une notion *subjective* utilisée pour structurer les programmes. On s'intéresse alors à la définition d'effets *sur mesure*.

calculatrice

Dans `calc.ml`, on peut considérer que chaque fonction `evalm` est un programme *avec effets*.

Chaque monade nous donne une notion de calcul avec des *effets sur mesure*.

raisonnement équationnel ?

Avec du code pur, on a :

```
let x = e1 in                    ≡    let y = e2 in
let y = e2 in                         let x = e1 in
e{x,y}                                e{x,y}
```

Avec un calcul monadique, ça dépend de la monade utilisée :

```
d1 >>= fun x ->                  ≃?   d2 >>= fun y ->
d2 >>= fun y ->                       d1 >>= fun x ->
e{x,y}                                e{x,y}
```

Valide pour `ErrorM`, `CountM` et `EnvM` mais pas `RngM`

(NB : valide pour `CountM` car `(+)` est commutatif)

raisonnement équationnel

Code pur :

```
let x = e1 in
let y = e1 in
e{x,y}
```

$\equiv$

```
let x = e1 in
e[y := x]{x}
```

Code monadique :

```
d1 >>= fun x ->
d1 >>= fun y ->
e{x,y}
```

$\simeq$?

```
d1 >>= fun x ->
e[y := x]{x}
```

Valide pour `ErrorM` et `EnvM`, mais pas `CountM` ou `RngM`.

raisonnement équationnel

Code pur :

```
let _ = e1 in
e{}
```

$\equiv$

```
e{}
```

Code monadique :

```
e1 >>= fun _ ->
e{}
```

$\simeq$?

```
e{}
```

Valide pour `EnvM`, mais pas `ErrorM` ni `CountM` ou `RngM`.

deux visions

Quel lien entre ces deux visions des effets ?
(effets comme primitives vs programmation avec effets sur mesure)

L'exemple de la calculatrice nous montre qu'il est possible d'**encoder** des effets ($\rightarrow$ en écrivant des programmes à base de `>>=` et `return`).

pourquoi encoder les effets ?

- **comme technique de programmation pour structurer les programmes**
- parce que son langage de programmation favori n'est pas doté de tel ou tel effet
- pour définir une sémantique de langage avec effets dans un langage mathématique pur

Traduction monadique

Dans un langage avec effets primitifs, on pourrait distinguer :

- le type des fonctions **pures** : $\alpha \to \beta$
- le type des fonctions **utilisant des effets** : $\alpha \rightsquigarrow \beta$

(En OCaml, tous les types de fonctions correspondent à $\alpha \rightsquigarrow \beta$.)

Une alternative est de décomposer les fonctions avec effet en :
fonction pure + calcul

$$\alpha \rightsquigarrow \beta \qquad \text{devient} \qquad \alpha \to \beta\ t$$

où $\beta\ t$ est une monade

On peut en faire une traduction systématique, en transformant les programmes d'un langage avec effets primitifs en programmes monadiques dans un langage pur.

(Et si la traduction cible des objets mathématiques, alors cela revient à donner une sémantique dénotationnelle pour le langage...)

C'est surtout utile d'un point de vue *théorique*.

Supponsons un langage λ^{eff} équipé d'un effet primitif.

Si l'effet primitif de λ^{eff} peut être représenté par une monade t, on peut transformer tout terme $e : \alpha$ de λ^{eff} en un terme $[\![e]\!] : [\![\alpha]\!]\ t$ d'un langage pur.

traduction monadique

la traduction rend explicite les effets monadiques et l'ordre d'évaluation :

$$
\begin{array}{rcl}
[\![\mathit{cst}]\!] & = & \texttt{return}\ \mathit{cst} \\
[\![\lambda x.\, u]\!] & = & \texttt{return}\ (\lambda x.\, [\![u]\!]) \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![u\ v]\!] & = & [\![u]\!] >\!\!>\!\!= \lambda u'. \\
 & & [\![v]\!] >\!\!>\!\!= \lambda v'. \\
 & & u'\ v' \\
[\![\mathrm{let}\ x\ =\ u\ \mathrm{in}\ v]\!] & = & [\![u]\!] >\!\!>\!\!= \lambda x.\, [\![v]\!]
\end{array}
$$

et on suppose donnée une opération dans la monade pour chaque primitive d'effet

traduction monadique

effet sur les types :

$$
\begin{aligned}
[\![\vdash^{\mathrm{eff}} e : \alpha]\!] &= \vdash [\![e]\!] : [\![\alpha]\!]\ t \\
[\![\mathbb{N}]\!] &= \mathbb{N} \quad \text{(pareil pour les autres types de base)} \\
[\![\alpha \to \beta]\!] &= [\![\alpha]\!] \to [\![\beta]\!]\ t
\end{aligned}
$$

Transformateurs de monades

comment combiner des monades ?

On peut vouloir combiner les effets de plusieurs monades.

Idée : implémenter des *transformateurs de monade*, qui combinent l'effet d'une monade choisie à une monade arbitraire.

Par exemple (voir `calc.ml`), transformateur pour la monade option :

```
module OptionT : (M: MONAD) -> sig
  include MONAD
  val fail : 'a t
  val lift : 'a M.t -> 'a t
end
```

NB : Avec plusieurs transformateurs, il faut alors faire attention à l'ordre des fonctions `lift`...

Effets : récapitulons

effets typés vs. non typés

Un effet peut être :

- “typé” : pris en compte par le système de types
- “non typé” : son utilisation ne se voit pas dans les types

- lorsque l'on écrit du code monadique, les effets sont *typés*
- en OCaml et Haskell, la non-terminaison et les exceptions sont des effets *non-typés*

effets primitifs vs. définis par l'utilisateur

Les monades correspondent à des effets "définis par l'utilisateur".

À l'inverse, les références, exceptions, etc. en OCaml sont des effets *primitifs*.

monades

les monades sont une technique utile pour structurer du code fonctionnel

Lorsque l'on programme, il est toujours utile de se demander :

≪ ce code pourrait-il être réécrit comme un *programme monadique* plus simple ? ≫

quitte à définir soi-même la bonne monade dont on a besoin !