Programmation Fonctionnelle Avancée

Séance 7 : Monades Reader et State, Map

Alexandros Singh

Université Paris 8

24 novembre 2023

Supposons que nous voulions composer des fonctions qui dépendent de valeurs existant dans un “environnement partagé”. Par exemple, considérons le dictionnaire suivant et quelques fonctions pour le manipuler :

```
type context = {name : string; surname : string; age : int}
let get_full_name d = d.name ^ " " ^ d.surname
let get_age d = string_of_int d.age
let make_pretty n a =
    "Hi " ^ n ^ ", you are : " ^ a ^ " years old."
let tell_age y d =
    "In " ^ (string_of_int y) ^ " year(s) you'll be " ^
    (string_of_int (d.age + y) ^ "!")
```

Reader - intro

Nous pouvons “composer” ces fonctions en passant explicitement l'environnement partagé à chaque fonction qui en a besoin :

```
let greet d =
    let name = get_full_name d in
    let age = get_age d in
    let ps = make_pretty name age in
    print_endline ps
```

```
# greet {name = "Alex"; surname = "Singh"; age = 29};;
Hi Alex Singh, you are : 29 years old.
- : unit = ()
```

Une façon plus propre de gérer un tel environnement partagé est d'utiliser la monade suivante :

```
module ContextReader : Monad with type 'a t = (context -> 'a) = struct
    type 'a t = (context -> 'a)
    let return x = fun _ -> x
    let ( >>= ) x f = fun w -> f (x w) w
end
```

Exercice : vérifiez que vous comprenez la définition de cette monade.
Exercice : vérifiez les trois lois des monades pour cette monade.

Il est facile d'adapter les fonctions précédentes au contexte monadique. Les deux premiers ne changent pas du tout (pourquoi ?) :

```
1 let get_full_name d = d.name ^ " " ^ d.surname
2 let get_age d = string_of_int d.age
```

Alors que `make_pretty` devient :

Il est facile d'adapter les fonctions précédentes au contexte monadique. Les deux premiers ne changent pas du tout (pourquoi ?) :

```
let get_full_name d = d.name ^ " " ^ d.surname
let get_age d = string_of_int d.age
```

Alors que `make_pretty` devient :

```
let make_pretty n a = return ("Hi " ^ n ^ ", you are : " ^ a ^
                              " years old.")
```

Reader - intro

La fonction `greet` peut alors être réécrite comme suit :

```
let greet =
    let* name = get_full_name in
    let* age = get_age in
    let* ps = make_pretty name age in
    return (print_endline ps)
```

Et il fonctionne exactement de la même manière :

```
# greet;; (*greet est une valeur monadique*)
- : unit ContextReader.t = <fun>
# (* c'est-à-dire une fonction qui prend un contexte ! *);;
# greet {name = "Alex"; surname = "Singh"; id = 100};;
Hi Alex Singh, your id is: 100.
- : unit = ()
```

Définition de la monade Reader

La monade `ContextReader` est un exemple d'une monade Reader qui, de manière plus générale et abstraite, peut être définie comme une structure ayant la signature suivante :

```
type ('a, 'c) readerT = Reader of ('c -> 'a)

module type ReaderMonad = sig
    include Monad
    type c
    val runReader: 'a t -> c -> 'a
    (* Exécute un calcul dans un environnement modifié. *)
    val local: (c -> c) -> 'a t -> 'a t
    (* Récupère l'environnement partagé. *)
    val ask : c t
end
```

Foncteur pour Reader

En définissant un module pour contenir notre type d'"environnement partagé", nous pouvons créer un foncteur qui nous donne des instances de la monade Reader :

```
module type Context = sig type c end
module MakeReader (C : Context) : (ReaderMonad with type c = C.c)
with type 'a t = ('a, C.c) readerT = struct
  type c = C.c
  type 'a t = ('a, c) readerT
  let runReader = function | Reader r -> r
  let ( >>= ) x f = Reader (fun w -> runReader (f (runReader x w)) w)
  let return x = Reader (fun _ -> x)
  let local f m = Reader (fun w -> runReader m (f w))
  let ask = Reader (fun env -> env)
end
```

Les fonctions précédentes peuvent être adaptées comme suit :

```
let get_full_name = Reader (fun d -> d.name ^ " " ^ d.surname)
let get_age = Reader (fun d -> string_of_int d.age)
let make_pretty n a = return ("Hi " ^ n ^ ", you are : " ^ a ^
                              " years old.")
```

Tandis que `greet` reste inchangé et peut être exécuté comme suit :

```
# runReader greet {name = "Alex"; surname = "Singh"; age = 29};;
Hi Alexandros Singh, you are : 29 years old.
- : unit = ()
```

Monade Reader

Comme nous l'avons déjà mentionné, `local` exécute des calculs dans un environnement modifié :

```
let local_example =
    let* name = local (fun d -> {d with name = "Alexandros"})
                      get_full_name in
    let* age = get_age in
    let* ps = make_pretty name age in
    return (print_endline ps)
```

Devinez le résultat :

```
# runReader greet {name = "Alex"; surname = "Singh"; age = 29};;
???
```

Monade Reader

Comme nous l'avons déjà mentionné, `local` exécute des calculs dans un environnement modifié :

```
let local_example =
    let* name = local (fun d -> {d with name = "Alexandros"})
                      get_full_name in
    let* age = get_age in
    let* ps = make_pretty name age in
    return (print_endline ps)
```

Devinez le résultat :

```
# runReader greet {name = "Alex"; surname = "Singh"; age = 29};;
Hi Alexandros Singh, you are : 29 years old.
- : unit = ()
```

Et cette fonction, qu'est-ce qu'elle donne comme résultat ?

```
let local_example_2 =
   let* name = local (fun d -> {d with name = "Alexandros"})
                     get_full_name in
   let* name_2 = get_full_name in
   return (name ^ " " ^ name_2)
```

```
# runReader local_example_2 {name = "Alex"; surname = "Singh";
                             age = 29};;
???
```

State - intro

Mais qu'en est-il si nous voulons apporter des modifications à l'environnement qui soient visibles pour toutes les autres fonctions qui suivent ? Par exemple, pour manipuler une pile, nous pouvons "composer" les fonctions du module Lifo que nous avons implémenté dans la session 3 :

```
let manip_stack =
    let open Lifo in
    let p1 = empty in
    let p2 = push 1 p1 in
    let p3 = push 2 p2 in
    peek p3
```

```
# manip_stack;;
- : int option = Some 2
```

Définition de la monade State

Fonctionne bien, mais pourrait être rendu plus propre avec... vous l'avez deviné la monade `State` ! Avant de le définir, définissons d'abord un type dont le constructeur unique englobe une fonction qui prend un état et renvoie une paire composée d'une valeur et d'un nouvel état.

```
type ('a, 'b) stateT = State of ('b -> 'a * 'b)
```

Définition de la monade State

Nous pouvons maintenant définir notre monade :

```
module type StateMonad = sig
    include Monad
    type s
    val runState: 'a t -> s -> 'a * s
    (* Retourne l'état contenu dans l'intérieur de la monade. *)
    val get: s t
    (* Remplacer l'état à l'intérieur de la monade. *)
    val put : s -> unit t
end
```

Foncteur pour State

Et voici le foncteur qui permet de créer des instances de cette monade :

```
module type State = sig type s end
module MakeState (S : State) : (StateMonad with type s = S.s)
with type 'a t = ('a, S.s) stateT = struct
  type s = S.s
  type 'a t = ('a, S.s) stateT
  let runState = function | State f -> f
  let ( >>= ) m f = State (fun s ->
                                let (a, newState) = runState m s in
                                let b = f a
                                in runState b newState)
  let return x = State (fun s -> (x,s))
  let get = State (fun s -> (s,s))
  let put new_s = State (fun _ -> ((), new_s))
end
```

Map

Enfin, puisque vous en aurez besoin pour le travail, voici une brève introduction au module Map :

- Le module gère des collections de clés et de valeurs créés à l'aide d'un foncteur appelé `Map.Make`.

Map

Enfin, puisque vous en aurez besoin pour le travail, voici une brève introduction au module Map :

- Le module gère des collections de clés et de valeurs créés à l'aide d'un foncteur appelé `Map.Make`.
- `Map.Make` crée des collections dont les clés appartiennent à un type dont les valeurs peuvent être comparées.

Ces types peuvent être intégrés dans des modules dont la signature est :

```
module type OrderedType = sig
    type t
    val compare : t -> t -> int
end
```

Compare définit un ordre total sur les valeurs du type `t` : `compare e1 e2` est 0 si ses entrées sont égales, strictement négatif si la première est plus petite que la seconde, et strictement positif si elle est plus grande. Nous utiliserons généralement la fonction intégrée `compare` pour l'implémenter.

Map

Une map simple contenant des chaînes de caractères comme clés et des entiers comme valeurs peut être créée et utilisée comme suit :

```
module StringOrd : Map.OrderedType with type t = string = struct
    type t = string
    let compare = compare
end
module StrMap = Map.Make(StringOrd)
```

```
# let example =
let open StrMap in
empty |> add "first" 1 |> add "second" 10 |> find "first";;
val example : int = 1
```

Map

Une map simple contenant des chaînes de caractères comme clés et des entiers comme valeurs peut être créée et utilisée comme suit :

```
module StringOrd : Map.OrderedType with type t = string = struct
    type t = string
    let compare = compare
end
module StrMap = Map.Make(StringOrd)
```

```
# let example2 =
let open StrMap in
empty |> add "first" 1 |> add "second" 10 |> find "second";;
val example2 : int = 10
```